HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

DIPLOMARBEIT

Herr
Martin Dörner

Modellierung und Simulation von verteilten Anwendungen auf drahtlosen Sensornetzwerken mit SciCos und OMNeT++

2011

Fakultät **Elektrotechnik und Informationstechnik**

DIPLOMARBEIT

Modellierung und Simulation von verteilten Anwendungen auf drahtlosen Sensornetzwerken mit SciCos und OMNeT++

Autor:
Martin Dörner

Studiengang:
Elektrotechnik/Automatisierungstechnik

Seminargruppe:
ET06wA1

Erstprüfer:
Prof. Dr.-Ing. Dietmar Römer

Zweitprüfer:
Dr.-Ing. Michael Galetzka

Mittweida, Januar 2011

Bibliografische Angaben

Dörner, Martin: Modellierung und Simulation von verteilten Anwendungen auf drahtlosen Sensornetzwerken mit SciCos und OMNeT++, 71 Seiten, 32 Abbildungen, Hochschule Mittweida (FH), Fakultät Elektrotechnik und Informationstechnik

Diplomarbeit, 2011

Referat

Drahtlose Sensornetzwerke werden in Zukunft immer häufiger zum Einsatz kommen und somit auch drahtgebundene Lösungen ersetzen. Die Anwendungsgebiete reichen von der Automatisierungstechnik, über die Gebäudeautomatisierung bis hin zur Medizintechnik. Dabei können verteilte Regelungen entstehen, die eine Vielzahl an Sensoren und Aktoren besitzen.

Um solch eine Komplexität im Entwurf und der Implementierung zu überschauen, wird in dieser Diplomarbeit die Realisierbarkeit einer Co-Simulation von verteilten Regelungen mit Hilfe von SciCos und OMNeT++, im Allgemeinen und anhand eines konstruierten Anwendungsbeispiels, aufgezeigt.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

Listings

III. Abkürzungsverzeichnis

CSMA	Carrier Sense Multiple Access
DLL	Dynamic Link Library
GUI	Graphical User Interface
HART	Highway Addressable Remote Transducer
IIS	Institut für Integrierte Schaltungen
IPv6	Internet Protocol version 6
MANET	Mobile Ad-hoc Network
MPLS	Multiprotocol Label Switching
NCS	Network Control System
NS2	Network Simulator 2
ODE	Ordinary Differential Equation
OSI	Open Systems Interconnection
OSPFv3	Open Shortest Path First version 3
TCP	Transmission Control Protocol
VoIP	Voice over Internet Protocol
WLAN	Wireless Local Area Network
WNCS	Wireless Network Control System
WSN	Wireless Sensor Network
XML	Extensible Markup Language

IV. Vorwort

Die vorliegende Diplomarbeit wurde in der Zeit von August bis Dezember 2010 am Fraunhofer Institut für Integrierte Schaltungen in Dresden unter Leitung von Dr. Michael Galetzka angefertigt. Vorausgegangen war eine reichlich dreimonatige Tätigkeit als studentische Hilfskraft zur Einarbeitung in die genutzten Softwareumgebungen und zur Validierung der Durchführbarkeit der Diplomarbeit.

1 Einleitung

Die Bedeutung von drahtlosen Sensornetzwerken (Wireless Sensor Networks - WSN) in der Industrie- und Gebäudeautomatisierung nimmt immer mehr zu. WLAN (Wireless Local Area Network) ist in vielen Bereichen der Industrie bereits akzeptiert und integriert. IEEE 802.15.4 hat sich jedoch noch nicht etabliert, obwohl die Anwendungsgebiete hier sehr umfangreich sind.

Mit IEEE 802.15.4 ist es möglich, Sensoren, Aktoren und andere Feldgeräte drahtlos miteinander zu verbinden. Es definiert die untersten zwei Schichten des OSI-Modells (Open Systems Interconnection), die Bitübertragungs- und Sicherungsschicht. Die Funktionalitäten der höheren Schichten übernehmen andere Standards, wie zum Beispiel ZigBee oder WirelessHART (Highway Addressable Remote Transducer). Die vernetzten Geräte lassen sich problemlos konfigurieren und auch schnell umkonfigurieren. Dadurch wird IEEE 802.15.4 sowohl für die Industrie als auch für die Heimautomatisierung interessant.

Im IEEE 802.15.4-Standard wird großer Wert auf einen geringen Energiebedarf gelegt. So soll es möglich sein, dass Knoten, die lediglich mit einer Batterie ausgerüstet sind, über Jahre wartungsfrei betrieben werden können. Dadurch kann eine Automatisierung auch an schwer zugänglichen Bereichen ermöglicht werden. Es können aber auch Netzwerkknoten ohne konstante Stromversorgung realisiert werden. So genannte Harvester-Knoten beziehen ihre Energie aus ihrer Umgebung, wobei eine Vielzahl an Energiequellen, wie Solar- oder Piezo-Module, einsetzbar sind.

Bei IEEE 802.15.4 geht es nicht um Geschwindigkeit und Datenraten, sondern vielmehr um eine hohe Flexibilität. Dadurch sind nicht nur einfache Steuerungsapplikationen, sondern auch komplexe Regelungen denkbar, die aus mehreren, über eine Anlage oder Gebäude verteilten Netzwerkknoten, bestehen.

Am Fraunhofer Institut für Integrierte Schaltungen (IIS) in Dresden wird eine Entwurfs-

unterstützung für drahtlose Sensornetzwerke mit einer großen Anzahl an Netzwerkknoten entwickelt. Solche komplexen Netzwerke mit einem hohen Verteilungsgrad sind mit herkömmlichen Entwicklungsmechanismen nicht mehr beherrschbar, weswegen die Notwendigkeit einer neuen Entwurfstechnologie besteht.

1.1 Motivation

Bei einer verteilten Regelungsapplikation in einem drahtlosen Netzwerk sind zwei Aspekte von großer Bedeutung, die bei einer praktischen Umsetzung beachtet werden müssen:

- Die physikalisch korrekte Modellierung der Regelung sowie der Regelungsumgebung.
- Die Effekte, die innerhalb einer Regelung durch ein drahtloses Netzwerk verursacht werden.

Deshalb ist es notwendig, zwei unterschiedlich spezialisierte Programme für eine Reglersimulation zu verwenden. Eines für die physikalische Modellierung und eines für die Simulation des drahtlosen Netzwerkes. Man spricht in diesem Zusammenhang auch von einer Co-Simulation.

So können die Auswirkungen von typischen Eigenschaften eines drahtlosen Netzwerkes, wie Verzögerungen, Paketverluste oder komplette Knotenausfälle auf eine verteilte Regelungsanwendung untersucht werden. Darüber hinaus bietet eine solche Co-Simulation neue Möglichkeiten, wie zum Beispiel die Simulation von Harvester-Knoten, die innerhalb des Netzwerkes nicht immer erreichbar sind.

Wie robust ist eine Regelung, deren Sensoren nur einmal pro Stunde ihre Werte übermitteln können? Was passiert mit der Regelung, wenn ein Sensor oder Aktor überhaupt nicht mehr erreichbar ist? Diese Fragestellungen sind die wesentliche Motivation dieser Arbeit.

Eine weiterführende Fragestellung bietet die Simulation eines drahtlosen Netzwerkes, in dem eine Vielzahl von unterschiedlichen, sich unbekannten Anwendungen, realisiert sind. Wie reagiert eine drahtlose Alarmanlage, wenn beispielsweise die Rollladensteuerung des Hauses mit voller Leistung sendet, um die Sonneneinstrahlung zu regeln? Wie können solche, im Einzelnen robuste Anwendungen, sich die Ressourcen eines drahtlosen Netzwerkes teilen ohne ihre Funktionalität zu verlieren?

Außerdem werden die Probleme, die beim Umsetzen eines zeitkontinuierlichen Regelungsmodells in ein ereignisgesteuertes Netzwerkmodell entstehen, betrachtet.

Solche Fragen sollen mit dieser Co-Simulation beantwortet werden können, um eine verteilte Regelung optimal an seinen Einsatzort beziehungsweise Einsatzzweck anzupassen.

In dieser Arbeit wird zum Modellieren der physikalischen Eigenschaften des Modells SciCos, ein Simulink ähnlicher Blockeditor, verwendet. Für die Simulation des drahtlosen Netzwerkes wird OMNeT++, ein ereignisgesteuerter Netzwerksimulator, genutzt. Mithilfe des in SciCos integrierten Code-Generators wird es möglich sein, das modellierte Regelungsmodell in den Netzwerksimulator zu übertragen.

1.2 Zielsetzung

Ziel der Diplomarbeit ist es, einen Weg aufzubereiten, wie Systemmodelle verteilter Regelungsanwendungen in drahtlosen Netzwerken (Wireless Networked Control Systems - WNCS) in Modelle auf Applikationsebene der einzelnen Knoten im Netzwerksimulator transformiert werden können. Somit sollen detaillierte Einflüsse der Kommunikation im WSN auf die WNCS im Netzwerksimulator untersucht werden. Dabei werden im zweiten Kapitel Arbeiten betrachtet, die sich mit den Problemen eines drahtlosen Sensornetzwerkes befassen. Zusätzlich werden Projekte vorgestellt, die Co-Simulationen von Regelungsprozessen bereits realisiert haben. Dabei werden die möglichen Programme vorgestellt, die für diese Art von Co-Simulationen geeignet sind.

Im dritten Kapitel werden die Programme SciCos und OMNeT++ vorgestellt, die für die

Durchführung der Simulation genutzt werden. Hier wird der SciCos Blockeditor erläutert und es wird auf die Funktionsweise der automatischen Code Generierung eingegangen. Anhand von OMNeT++ werden die Besonderheiten einer ereignisgesteuerten Netzwerksimulation aufgezeigt.

Im vierten Kapitel wird die Vorgehensweise beim Modellieren einer verteilten Regelungsapplikation in SciCos unter Beachtung der späteren Code-Generierung, aufgezeigt. Es wird erläutert, wie der generierte Code zu einer Anwendung auf einen Netzwerkknoten in OMNeT++, genutzt wird.

Im fünften Kapitel wird dieser Ablauf anhand eines praktischen Beispiels verdeutlicht. Es werden die Simulationsergebnisse von SciCos und OMNeT++ verglichen und die Realisierbarkeit der modellierten Regelung abgeschätzt.

Abschließend soll aufgezeigt werden, was solch eine Co-Simulation in Zukunft leisten könnte. Zudem sollen notwendige Schritte aufgezeigt werden, um den Vorgang zu automatisieren und somit praktikabel zu gestalten.

2 Verwandte Arbeiten

Es gibt eine Vielzahl von wissenschaftlichen Arbeiten, die sich mit den neuen Möglichkeiten, aber auch mit den zusätzlichen Schwierigkeiten einer Regelungsanwendung über ein drahtloses Sensornetzwerk befassen. Einige Arbeiten über Sensornetzwerke und Co-Simulationen von drahtlosen Sensornetzwerken werden auf den folgenden Seiten vorgestellt.

2.1 Regelungsanwendungen über drahtlose Netzwerke

Regelkreise oder Steuerungen, bei denen die gesamte Kommunikation zwischen den einzelnen Komponenten, wie Sensoren, Aktoren und Controller, teilweise oder komplett über ein Netzwerk stattfindet, nennt man Networked Control Systems (NCS). Ist die Übertragung der Nutzdaten nicht drahtgebunden über einen Feldbus oder Ethernet realisiert, sondern über ein drahtloses Funknetzwerk, so spricht man von einem Wireless Networked Control System (WNCS).

Die Schwierigkeiten eines WNCS sind zum großen Teil schon in einem NCS zu finden. Wie in [1] beschrieben, gibt es drei grundlegende Störgrößen, die Leistung und Stabilität einer Regelung beeinflussen können:

1. Verzögerungen (Totzeiten)
2. Paketverluste
3. falsche Reihenfolge der Pakete beim Empfänger

Weitere Fehlerquellen entstehen bei der Nutzung eines WNCS. Um diese genauer zu betrachten, ist eine zusätzliche Differenzierung notwendig. So wird in [2] eine klare Unterscheidung von fest strukturierten Netzwerken, wie WLANs, bei denen die gesamte Kommunikation über einen Router auf zuvor definierten Wegen verläuft, und Ad-hoc-Netzwerken (MANET), bei denen Netzknoten direkt miteinander kommunizieren können, getroffen. Das macht solch ein Netzwerk flexibler, jedoch wird das Routing kompli-

zierter, da keine zentrale Station existiert.

In [3] werden neben den Fehlerquellen aus [1] noch weitere, die Regelung beeinflussende Faktoren benannt, die durch die Unberechenbarkeit eines WNCS und eines MANET entstehen.

- Änderungen der Netzstrukturen
- Ortsänderungen der Netzknoten
- Jitter[1]
- Routing

In [3] werden auch die Eigenschaften eines MANET genauer beleuchtet. Der Vorteil der Selbstorganisation und Wiederherstellung nach Fehlerfällen ermöglicht demnach eine Vielzahl von Einsatzgebieten:

- Militär
- Rettungseinsätze
- Montagehallen
- Erkundung schwer zugänglicher Gegenden
- Chirurgie
- Motorenprüfstände
- Luftüberwachung
- Unterhaltungsindustrie

Ein nicht zu vernachlässigender Faktor beim WNCS ist der Pfadverlust. Die Reichweite von IEEE 802.15.4 kann im Bereich von zehn bis 100 Meter stark schwanken [4], so dass nie sicher ist, ob ein Knoten ein gesendetes Paket empfangen kann. Der Einfluss von Wänden innerhalb eines Gebäudes, aber auch die Reichweite im Freien ist ein wichtiges Forschungsgebiet. So wurde in einer Diplomarbeit am Fraunhofer Institut für Integrierte Schaltungen in Dresden ein Kanalmodell entwickelt, das in der vorliegenden Diplomarbeit bei der Netzwerksimulation verwendet wird.

[1] Laufzeitvarianz bei der Übertragung eines Digitalsignals.

2.2 Programme und Tools für eine Co-Simulation

Die analytische Untersuchung an WSN ohne Simulationsumgebung ist nach [5] oftmals teuer und zeitaufwendig. Hinzu kommt, dass in einem WCNS die Simulation des Netzwerk eine entscheidende Rolle spielt. Doch oftmals reicht das nicht aus. Die Modellierung der physikalischen Eigenschaften einer Regelung und besonders auch der realen Welt, mit der eine Regelung interagiert, ist genauso wichtig. Deshalb werden die jeweiligen Eigenschaften eines Systemmodellierungs-Programmes und eines Netzwerksimulators zu einer gemeinsamen Simulation, einer so genannten Co-Simulation, vereint. Die Programme und Tools, die sich dafür anbieten, werden nachfolgend vorgestellt.

System-Modellierung

Simulink ist weit verbreitet und wird in der Regel zum Erzeugen von Modellen von physikalischen Systemen verwendet. Es ist ein Zusatzprodukt von MATLAB[2], bei dem mit Hilfe grafischer Blöcke eine hierarchische Modellierung erstellt werden kann. Dazu stehen zeitkontinuierliche und diskrete Schaltblöcke zur Verfügung [6].

Modelica ist eine objektorientierte Beschreibungssprache zum Erstellen von physikalischen Modellen. Zur Untersuchung eines physikalischen Modells wird in Modelica ein Translator verwendet, mit dem ein mathematisches Modell erzeugt wird, welches mittels eines Lösungsalgorithmus gelöst wird. Es existieren eine Reihe von grafischen Entwicklungsumgebungen für Modelica. Sie nutzen ähnlich wie Simulink grafische Objekte, die über Konnektoren miteinander verbunden werden [7].

SciCos ist ein eine Erweiterung zu ScicosLab, welches ein umfangreiches Software-Paket ist. Die Funktionalität ist analog zu MATLAB/Simulink, so dass sogar Konverter von MATLAB zu ScicosLab existieren [8]. Der Umfang an Modellierungs-Blöcken ist in SciCos jedoch deutlich niedriger, was der geringeren Popularität geschuldet ist. Es können jedoch eigene Blöcke programmiert werden. 2008 kam es zu einer Aufspaltung des Konsortiums, so dass zurzeit die fast identische Software SciLab/XCos existiert.

[2] Software zum numerischen Lösen von mathematischen Problemen.

Netzwerksimulatoren

NS2 (Network Simulator 2) ist ein diskreter eventbasierter Netzwerksimulator. NS2 erlaubt eine Simulation von TCP/IP, Routing und Multicast Protokollen über verdrahtete und drahtlose Netzwerke [9].

OPNET Modeler ist ein Tool aus der „OPNET Technologies suite“. Er beinhaltet eine große Auswahl an Protokollen und Technologien und enthält eine Entwicklungsumgebung mit der alle gängigen Netzwerktypen wie VoIP, TCP, OSPFv3, MPLS oder IPv6 simuliert werden können [10].

OMNeT++ ist wie NS2 ein diskreter eventbasierter Netzwerksimulator. Er ist für die akademische Nutzung frei verfügbar und besteht aus verschiedenen Modulen und Komponenten. Die gebräuchlichste Nutzung von OMNeT++ ist die Simulation von Computer Netzwerken. Durch die Erweiterung mit Hilfe von Frameworks[3] sind andere Netzwerkstrukturen ebenso realisierbar [11].

2.3 Aktuelle Projekte

2.3.1 Die TrueTime-Toolbox

In [2] wird die TrueTime-Toolbox für Simulink vorgestellt, die auch in anderen Projekten, wie zum Beispiel in [12] zur Anwendung kommt. TrueTime stellt eine Block-Bibliothek zur Verfügung (Abbildung 2.1 aus [13]), welche die für die Automatisierungstechnik gängigsten Protokolle unterstützt [14]. Dazu gehören unter Anderem:

- Ethernet
- CAN
- PROFINET
- WLAN
- Round Robin
- ZigBee

[3] Grundgerüst aus Modulen und Klassen für die Softwareentwicklung.

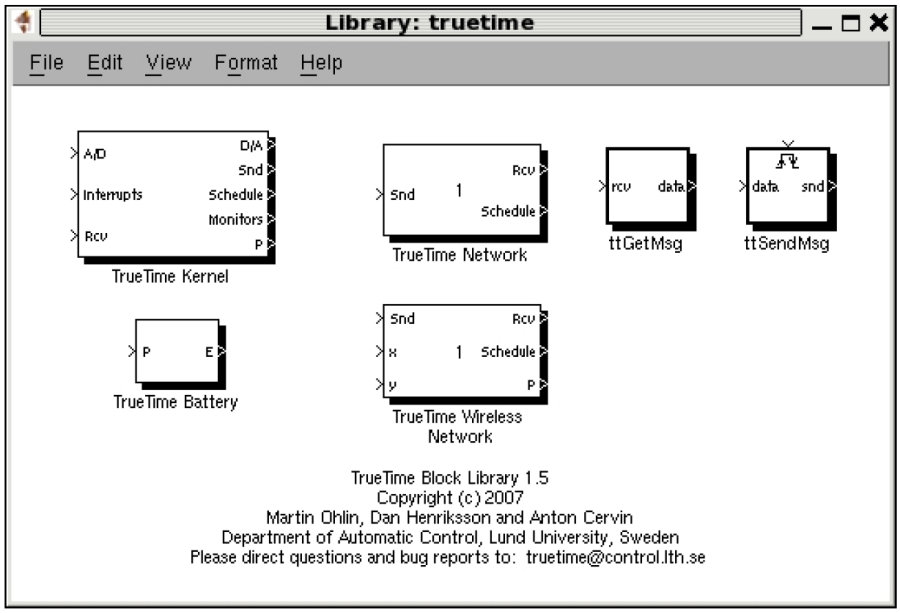


Abbildung 2.1: TrueTime Block-Bibliothek

Diese Netzwerk-Blöcke können mit den normalen Simulink-Blöcken verknüpft werden, wodurch eine einfache Co-Simulation mittels Simulink, ohne zusätzliche Software, realisiert wird. Die TrueTime-Modelle sind aber nur vereinfachte Netzwerk-Modelle. Für eine umfassendere Simulation ist ein spezieller Netzwerksimulator notwendig.

2.3.2 Die PiccSIM-Toolchain

Im Gegensatz zur TrueTime-Toolbox werden in PiccSIM laut [15] zwei unterschiedliche Programme verwendet. Das bevorzugte Anwendungsgebiet sehen die Entwickler in industriellen Regelungsapplikationen.

PiccSIM koppelt Simulink und NS2. Es ist eine Erweiterung für Simulink mit einer eigenen grafischen Benutzeroberfläche (GUI), über die eine Simulation konfiguriert wird. Für einen Simulationsvorgang werden zwei Rechner benötigt. Auf einem läuft MATLAB mit der PiccSIM-GUI, auf dem anderen der Netzwerksimulator NS2. Per Netzwerkkabel sind beide Rechner miteinander verbunden und die Kommunikation erfolgt über XML-Nachrichten. Mit Hilfe der Nachrichten werden Daten zur laufenden Simulation sowie zur Synchronisation ausgetauscht (Abbildung 2.2 aus [15]).

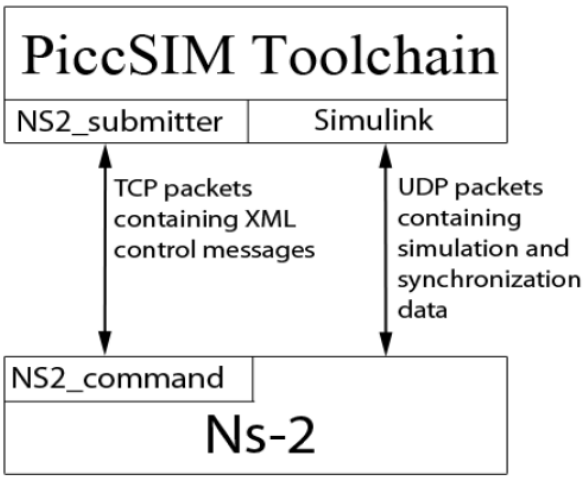


Abbildung 2.2: PiccSIM Architektur

Eine Simulation läuft folgendermaßen ab:

1. Simulink startet die Simulation.
2. Die aktuelle Konfiguration wird an NS2 übermittelt.
3. Simulink simuliert bis zur Zeit t_1.
4. Die aktuelle Simulationszeit t_1 wird an NS2 gesendet.
5. NS2 simuliert ebenfalls bis t_1 und sendet danach eine Bestätigung zurück.
6. Wenn Simulink diese Bestätigung erhält, wird bis zum nächsten Zeitpunkt t_2 simuliert.

Am Ende einer Simulation sendet NS2 das Simulationsergebnis an die Simulink-PiccSIM-GUI.

Zudem ist es möglich, mit Hilfe des Code-Generators von Simulink, aus dem modellierten Modell Quellcode zum Testen der Funktionalität auf Hardware, zu erzeugen. Dabei wird für alle Knoten der Regelung Quellcode erzeugt. Die möglichen Layouts für die Regelschleife sind in Abbildung 2.3 aus [15] dargestellt.

2.3.3 OPNET-Simulink

In [16] wird eine Co-Simulation von MATLAB/Simulink und OPNET präsentiert. Ein Anlagen-Modell und ein Netzwerk-Modell werden zusammen auf einem Rechner simu-

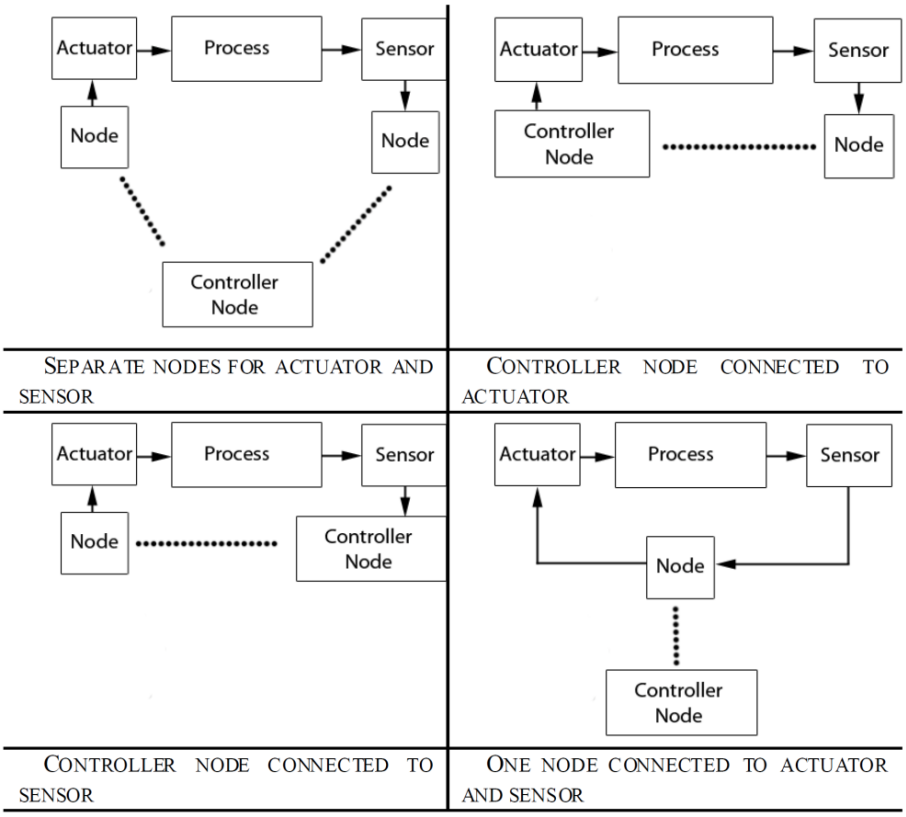


Abbildung 2.3: Mögliche Regelungs-Layouts

liert. Am Beispiel des Reglers für ein Doppel-Pendel-Simulink-Modell in Abbildung 2.4 aus [16] wird die Funktionsweise verdeutlicht. Links im Bild befindet sich ein Doppel-Pendel-Modell, rechts ein PID-Regler-Modell. Dazwischen ist ein MANET, über das die gesamte Kommunikation zwischen Sensoren und Aktoren am Pendel und PID-Regler stattfindet.

Die Synchronisation erfolgt analog zu der in PiccSIM. OPNET ist der Simulations-Master, der die Simulation frei gibt, worauf jeweils die einzelnen Simulationen starten. Dabei werden die Sensor- und Aktorwerte ausgetauscht. Die Gesamtverzögerung der Regelung ergibt sich dabei aus der Sensor-Regler- und Regler-Aktor-Verzögerung.

Neben der Co-Simulation wurden zusätzlich drei verschiedene MANET-Modelle in OPNET implementiert und verglichen:

Modell 1 nutzt einen Pfad-Verlust-Exponenten sowie eine Gauß'sche-Zufallsvariable.

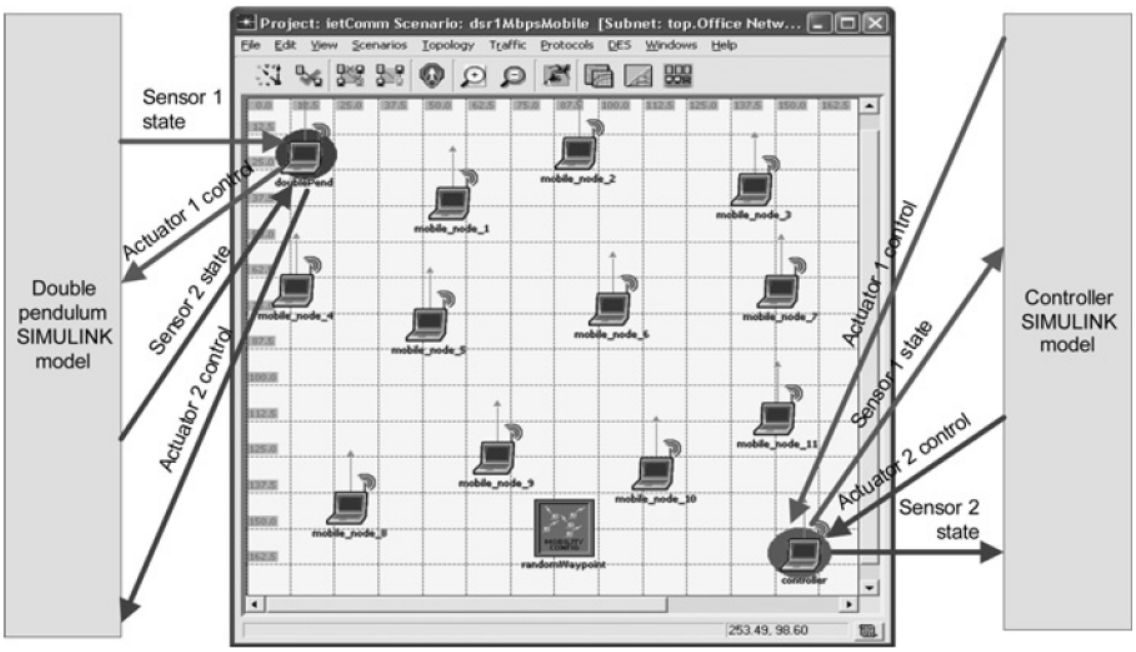


Abbildung 2.4: Co-Simulation von OPNET und Simulink

Modell 2 ist ein Zwei-Wege-Modell[4], das nur den Pfadverlust berücksichtigt.

Modell 3 ist ein idealisiertes Pfadverlust-Modell.

Diese Modelle wurden mit einer Messung im Freifeld verglichen, wobei lediglich das erste Modell den erhaltenen Messwerten nahe kam.

2.3.4 NS2-Modelica

Eine mögliche Co-Simulation von Modelica und NS2 ist in [17] dargestellt. Die technischen Schwierigkeiten sehen die Entwickler

- in der zu schaffenden Kommunikationsschnittstelle,
- in der Synchronisierung der Programme und
- beim Realisieren eines Master-Slave-Mechanismus zwischen einem Modelica-Modul und der dazugehörigen NS2-Simulation.

Der Kommunikations-Mechanismus zwischen den Programmen ist in Abbildung 2.5 dargestellt. Es ist zu erkennen, dass die Synchronisation nach dem gleichen Prinzip, wie in

[4] Der Weg zwischen Sender und Empfänger kann direkt oder über Reflexion am Boden zurückgelegt werden.

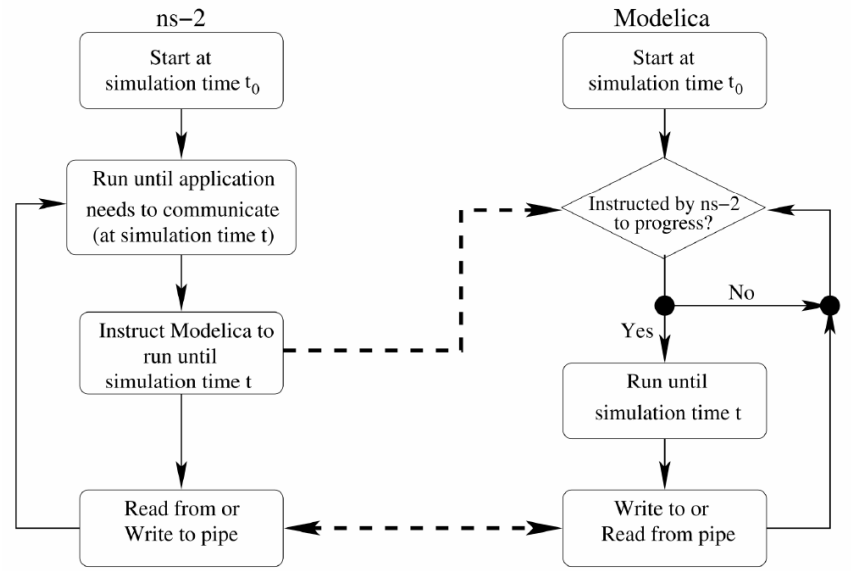


Abbildung 2.5: Die Kommunikation zwischen NS2 und Modelica

den beiden vorherigen Projekten realisiert wurde. Der Datenaustausch erfolgt innerhalb UNIX[5] mittels „pipes“[6] und ist in beide Richtungen möglich. Dabei ist NS2 der Master in dieser Co-Simulation.

2.3.5 Zusammenfassung

Es gibt eine Vielzahl von Co-Simulationen, die jedoch, ausgenommen der TrueTime-Toolbox, alle dem gleichen Prinzip folgen.

- Das Systemmodellierungs-Programm und der Netzwerksimulator laufen gleichzeitig und parallel ab.
- Es existiert eine Möglichkeit des Datenaustausches zwischen beiden Programmen.
- Der Datenaustausch realisiert eine Synchronisierung der Programme.
- Ein Programm fungiert als Master, ein weiteres als Slave.

[5] Ein Mehrbenutzer-Betriebssystem.
[6] Sie ermöglichen in UNIX die Kommunikation zwischen Prozessen [18].

Die Reglerstruktur kann in keiner der Co-Simulationen frei gewählt werden. Bei OPNET-Simulink aus Kapitel 2.3.3 wird lediglich der Weg zwischen Sensor und Regler mit einem drahtlosen Netzwerk überbrückt und bei der PiccSIM-Toolchain aus Kapitel 2.3.2 stehen lediglich vier Reglerstrukturen zur Wahl. Für den Entwurf einer verteilten Regelung sind solche Beschränkungen aber nicht erwünscht. Eine Co-Simulation von SciCos und OMNeT++ soll in dieser Hinsicht keinerlei Einschränkungen unterworfen sein.

3 Die verwendeten Simulatoren

Für diese Arbeit wurde zur Modellierung und Simulation der verteilten Regelungsanwendung SciCos verwendet. Zur Simulation des drahtlosen Sensornetzwerkes kam OMNeT++ zum Einsatz. OMNeT++ wird am Fraunhofer IIS in Dresden für die Entwicklung einer IEEE 802.15.4 Entwurfsunterstützung genutzt. Deshalb stand OMNeT++ trotzt der anderen verfügbaren Netzwerksimulatoren, schon vor Beginn der Arbeit als Zielplattform fest. Für das Systemmodellierungs-Programm ist die entscheidende Voraussetzung ein integrierter Code-Generator. Neben SciCos enthält auch Simulink einen Code-Generator. Da SciCos aber schon in anderen Projekten am Fraunhofer IIS erfolgreich zum Einsatz kam, wurde es für die Systemmodellierung verwendet.

3.1 SciCos

Der Modellsimulator SciCos ist ein grafischer Editor, mit dem dynamische Systeme modelliert und simuliert werden können. Der Nutzer kann durch die Verwendung verschiedener Blöcke aus der Bibliothek oder durch selbst entwickelte Blöcke Block-Diagramme erstellen und in ausführbaren Code kompilieren. Eine Modelica-Erweiterung erlaubt zusätzlich die Generierung von Modellen von elektrischen und hydraulischen Systemen [19]; [20]. In dieser Arbeit wird mit der SciCos-Version 4.3 gearbeitet. Parallel dazu war auch die Version 4.4b verfügbar, die jedoch aufgrund des Beta-Status vermieden wurde. In [19] werden die Anwendungsgebiete und Funktionen von SciCos genannt:

Anwendungsgebiete:

- Signalverarbeitung
- Regelungstechnik
- Warteschlangen-Verwaltung
- Modellierung von physikalischen und biologischen Systemen

Funktionen:

- Grafisches Modellieren, Kompilieren und Simulieren von dynamischen Systemen
- Kombination von kontinuierlichem und diskretem Zeitverhalten
- Umfangreiche Block-Bibliothek
- Entwicklung von eigenen Blöcken in C oder FORTRAN
- Erstellung von Super-Blöcken aus Teilkomponenten
- C-Code-Generierung aus Super-Blöcken
- Manuelle Wahl eines Solvers[7]

3.1.1 SciCos Beispiel

SciCos ist eine SciLab Toolbox und wird aus SciLab heraus gestartet. Nach dem starten öffnet sich ein leeres Fenster, in dem das Blockdiagramm erstellt wird. In diesem Beispiel wird ein einfacher PID-Regler entworfen (Abbildung 3.1). Mit Hilfe der Palette (Abbildung 3.2) können die dafür benötigten Blöcke per „Drag&Drop“ in das SciCos-Fenster gezogen werden.

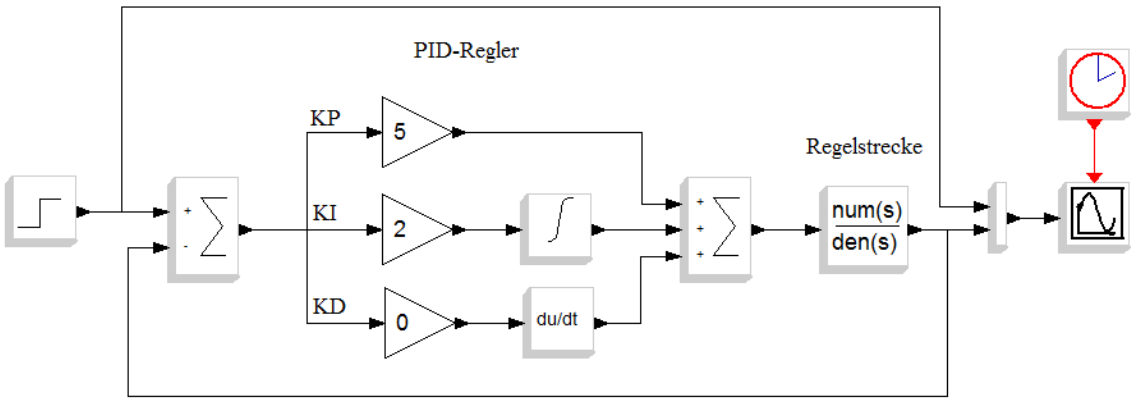


Abbildung 3.1: PID-Regler mit SciCos

Die Ein- und Ausgänge der Blöcke werden über Konnektoren miteinander verbunden. Konnektoren können Signalpfade (schwarz, horizontal) und Clock-Pfade (rot, vertikal) sein. Zeitkontinuierliche Blöcke, wie der Integral-Block, werden nicht getaktet und haben demnach keinen Clock-Eingang. Andere Blöcke, wie das Oszilloskop benötigen jedoch einen Takt, der im Menüfenster des Blocks eingestellt werden kann.

[7] Software zum Lösen von Differentialgleichungen.

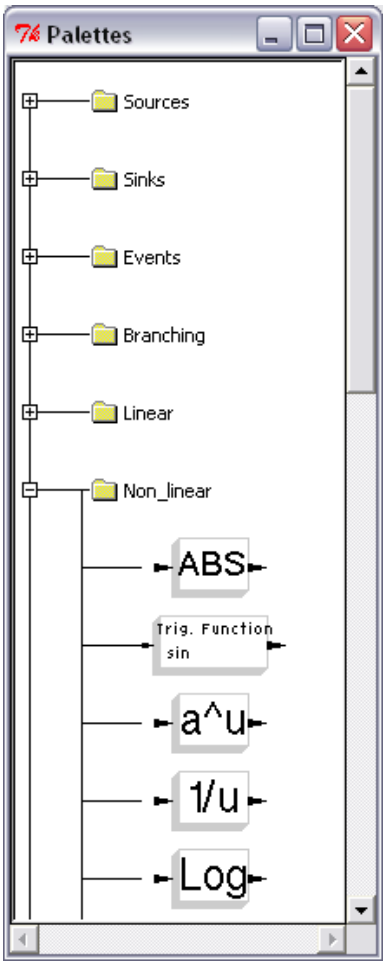


Abbildung 3.2: SciCos-Palette

Die kontinuierlichen Blöcke ohne Takteingang sind allerdings nicht ständig aktiv. Sie werden nur aufgerufen, wenn dies notwendig ist. Dabei gibt es zwei Prinzipien (Vgl. Kapitel 8.2 und 8.3 aus [20]):

1. Das Regler Beispiel aus Abbildung 3.1 besteht, bis auf den Graph-Block, ausschließlich aus so genannten **„Always Active Blocks“**. Diese werden nur aktiviert, wenn der Graph einen Wert darstellen muss. Abhängig vom Takt des Graph-Blocks, werden die vorangestellten Blöcke aufgerufen, um einen Wert zu berechnen.

2. Die **Vererbung** funktioniert prinzipiell genau wie bei den **„Always Active Blocks“**, nur in die entgegengesetzte Signalrichtung. Ist eine Quelle getaktet, so erben die nachfolgenden kontinuierlichen Blöcke ihre Aktivierung über den Signalpfad (Abbildung 3.3).

In diesem Beispiel wird als Sollwert ein Einheitssprung auf den Regler gegeben. Die Signale des Sollwertsprungs und der Sprungantwort des PID-Reglers verlaufen durch einen Multiplexer (Mux). Dadurch können beide Signale auf einem Oszilloskop-Block (Abbildung 3.4) ausgegeben werden.

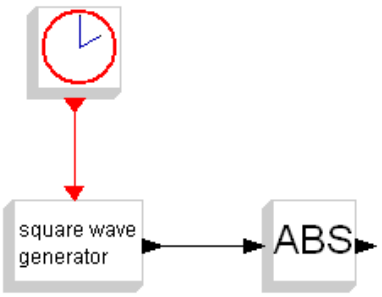


Abbildung 3.3: `ABS`-Block erbt Takt vom Generator

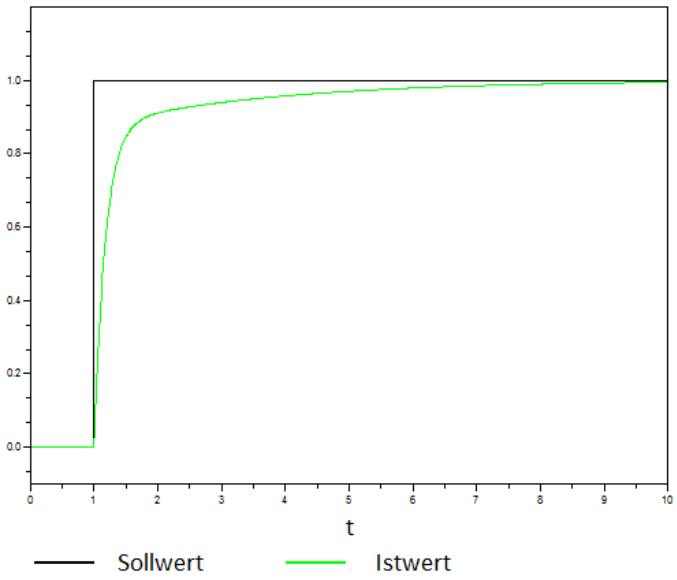


Abbildung 3.4: Einheitssprung mit Sprungantwort

3.1.2 Code-Generator

SciCos bietet einen Code-Generator an, der zum kreieren von C-Code für ein SciCos-Modell oder für Teilkomponenten eines Modells genutzt werden kann. Es gibt zwei Anwendungsgebiete für den Code-Generator:

1. Zum Verbessern der Leistung des Simulators:

   Wenn SciCos ein Modell simuliert, entsteht immer ein Overhead[8], der vermieden werden kann, wenn die Modelle in C-Code umgewandelt werden.

2. Zum Erstellen von eigenständigen Anwendungen:

   Dieser so genannte „Standalone-Code“ kann unabhängig von SciCos auf einem PC oder Mikrocontroller ausgeführt werden.

C-Code kann nur aus einem Super-Block generiert werden. Ein Super-Block kann aus der Palette entnommen und wie ein normales Modell erstellt werden. Es ist aber auch möglich, ein Teil eines Modells zu markieren, um daraus einen Super-Block zu bilden. Ein Super-Block verändert das Verhalten der Simulation in keiner Weise. Er stellt nur eine Art Gliederung des Modells in Ebenen dar. Große Modelle können dadurch übersichtlicher gestaltet werden. Am Beispiel des PID-Reglers wurde der Regler in Abbildung 3.5 in einen Super-Block gewandelt.

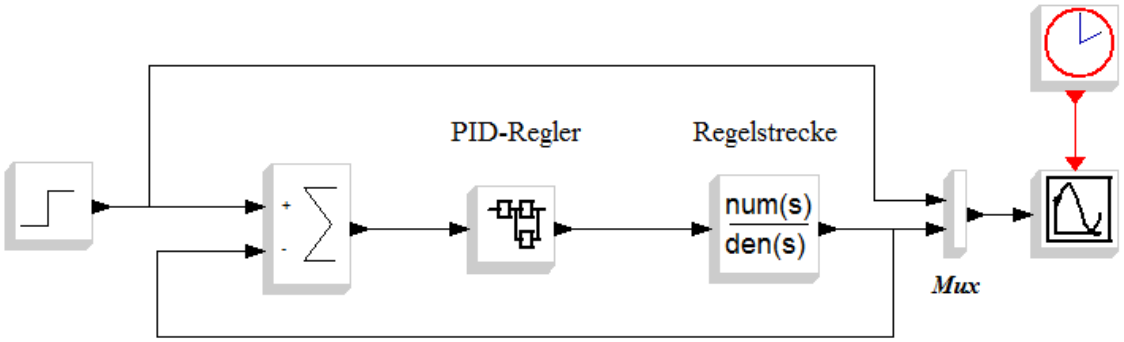


Abbildung 3.5: PID-Regler als Super-Block

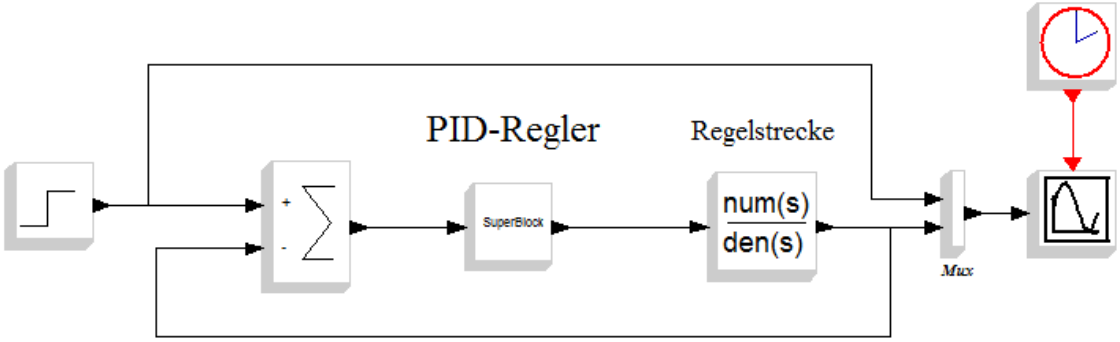


Abbildung 3.6: PID-Regler als C-Code

[8] Rechen-Mehraufwand, der keinen direkten Nutzen hat.

Das Erstellen von C-Code aus dem Super-Block ist nun über das Menü möglich. Es muss lediglich ein Name und der Speicherort des neuen Blocks eingetragen werden. In Abbildung 3.6 ist das Modell mit einem C-Code-Regler dargestellt. Zu beachten ist, dass dieser Vorgang nicht mehr rückgängig zu machen ist. Die Elemente des Super-Blocks sind jetzt nur noch als C-Code vorhanden; es können keine Änderungen mehr am Verhalten des Blocks vorgenommen werden. Die grafischen Menüs zum Einstellen der Blockparameter sind nicht mehr aufrufbar, etwaige Änderungen müssen im Quellcode dann direkt vorgenommen werden, was ohne fundiertes Wissen über den generierten Code kaum möglich ist. Am Verhalten der gesamten Simulation sollte sich jetzt nichts ändern. Jedoch kann es trotzdem zu ungewollten Beeinflussungen kommen, wie in Kapitel 12.4 aus [20] beschrieben ist:

1. Werden in einem Super-Block einzelne Blöcke von außen direkt aktiviert, andere jedoch vererben ihre Aktivierung, so kann nach der Code-Generation diese Unterscheidung nicht mehr vorgenommen werden. Der gesamte Block wird ab da an nur noch bei direkter Aktivierung aufgerufen. Die Vererbung findet nicht mehr statt.
2. Erbt beispielsweise ein Graph-Block seine Aktivierung von einem Super-Block, so kann es nach der Code-Generation ebenfalls zu Veränderungen kommen. Konnte der Super-Block noch differenziert betrachtet werden, so wird der Code-Block nur noch als Ganzes aufgerufen. Der Graph-Block würde dadurch bei jeder Aktivierung des Code-Blocks aufgerufen, was beim Super-Block zuvor nicht zwingend der Fall gewesen war.

3.2 OMNeT++

OMNeT++ ist ein diskreter ereignisorientierter Simulator. Ein Ereignis ist das Eintreffen einer Nachricht (Message) in einem Modul. Messages können von anderen Modulen, aber auch vom eigenen Modul stammen. Zwischen zwei Ereignissen bleibt das System, wie bei einem Zustandsautomat, unverändert. OMNeT++ ist objektorientiert und modular aufgebaut. Laut [21] ist der Simulator für viele Anwendungen und Bereiche nutzbar:

- Modellierung von drahtgebundenen und drahtlosen Netzwerken
- Modellierung von Protokollen
- Modellierung von Warteschlangen-Verwaltung
- Modellierung von Multiprozessoren und anderen verteilten Hardware-Systemen
- Validierung einer Hardware-Architektur
- Evaluierung der Performance von komplexen Software-Systemen
- Ganz allgemein, für das Modellieren und Simulieren eines Systems, bei dem das diskrete ereignisorientierte Verhalten anwendbar ist und welches in Kommunikationseinheiten, die untereinander Nachrichten austauschen, gegliedert werden kann.

OMNeT++ ist nicht auf konkrete Netzwerke spezialisiert, sondern stellt eine Infrastruktur und Werkzeuge bereit, um beliebige Event-Simulationen zu erzeugen. Ein fundamentaler Teil der Infrastruktur sind die Architekturkomponenten für die Simulationsmodelle. Diese Komponenten können vielfältig genutzt und für mehrere Modelle wiederverwendet werden [21].

Zur Simulation von WSN mit IEEE 802.15.4 stellt MiXiM weitere Komponenten bereit [22]. Dieses Framework bietet detaillierte Modelle zur Signalausbreitung, zum Stromverbrauch oder für die einzelnen Protokollschichten. Die Fraunhofer Simulationsplattform SuSAN erweitert unter Benutzung von MiXiM, OMNeT++ um weitere Funktionalitäten, insbesondere in Bezug auf die Simulation von IEEE 802.15.4-Netzwerken.

Ein Beispiel für eine eventbasierte Simulation, woran die Funktionsweise von OMNeT++ erläutert werden kann, ist „TicToc“ (Abbildung 3.7). Es ist eine einfache Netzwerksimulation, bei der mehrere Knoten versuchen, über Verbindungen mittels eines Routing-Verfahrens Messages zu senden.

Der Aufbau eines Moduls wird in NED-Files beschrieben, indem unter Anderem die Anzahl der Knoten und deren Verbindungen festgelegt werden. Das Verhalten eines Knotens wird in einer zugehörigen C++ Klasse implementiert. Klasse und NED-File bilden ein „simple Modul“. „Simple Moduls“ können über ihre Verbindungen Messages austauschen.

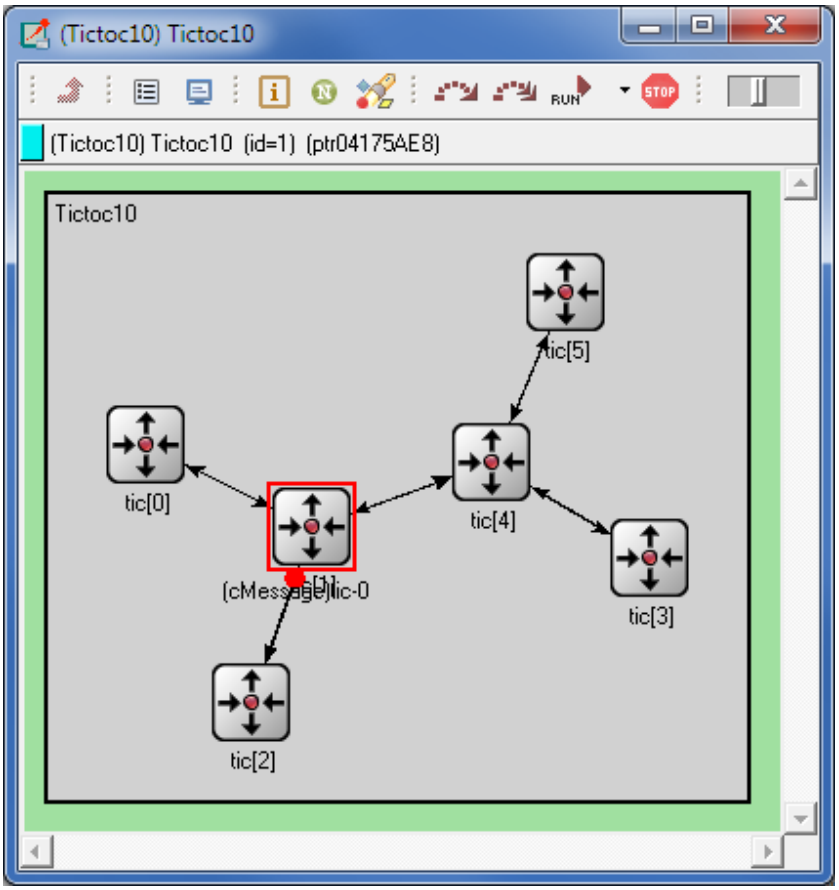


Abbildung 3.7: Ein OMNeT++-Netzwerk mit sechs Knoten

Das „Simple Modul“ für die Zielplattform des SciCos-Codes innerhalb OMNeT++, stellt MiXiM bereit. Darin soll mit Hilfe einer neuen C++-Klasse der Code eingebunden werden. MiXiM liefert unter anderem auch den „Connection Manager“, der die Verbindungen zwischen den Modulen koordiniert und ein „World“ Modul, das die physische Größe des drahtlosen Sensornetzwerkes festlegt.

4 Co-Simulation mit SciCos und OMNeT++

Bei der Erstellung eines Modells auf Applikationsebene in OMNeT++ mit Hilfe des SciCos Code-Generators hat die korrekte Vorgehensweise höchste Priorität. Deshalb wird in diesem Kapitel der Ablauf, beginnend bei der Erstellung eines SciCos-Modells, die anschließende Code-Generierung und schließlich die Integration des C-Codes in OMNeT++, behandelt. Zudem werden allgemeine Eigenschaften einer dezentralen Regelung, das grundsätzliche Zeitverhalten in SciCos und OMNeT++ und die entstehenden Totzeiten innerhalb eines IEEE 802.15.4-Netzwerkes erläutert.

4.1 Eigenschaften einer Regelung in WNCS

4.1.1 Dezentrale Regelung

Eine verteilte oder dezentrale Regelung ist dadurch gekennzeichnet, dass der Regler nicht zentral vorliegt, sondern auf mehrere Komponenten verteilt wurde. Am anschaulichsten kann dieses Verhalten bei der Realisierung einer Mehrgrößenregelung gezeigt werden. Bei einer Mehrgrößenregelung werden mehrere Größen geregelt, was mit einem Regler oder auch mit mehreren Reglern realisiert werden kann. Sind zwei oder mehr Regler im Einsatz spricht man bereits von einer verteilten Regelung.

Beispiele für Mehrgrößrenregelungen können sein:

- Eine Klimaregelung, bei der die Temperatur die relative Luftfeuchtigkeit beeinflusst.
- Die Regelung eines Industrieroboters, wobei die Position des „Tool Center Points“ von der Position, Kraft und Geschwindigkeit der einzelnen Teilachsen abhängt.
- Ein Dampfkessel, in dem die erzeugte Menge Wasserdampf von Temperatur und Druck des Wassers abhängig ist.

Dabei müssen die Regler nicht zwangsweise untereinander Informationen austauschen.

Die Kopplung entsteht meist durch die Regelstrecke. Beim Beispiel der Klimaregelung hat eine Änderung der Temperatur auch eine Änderung der relativen Luftfeuchte zur Folge. Wodurch die Änderung der Luftfeuchte ausgelöst wurde, ist für den Regler nicht von Bedeutung. Die Änderung wird wie eine Störgröße betrachtet. Beide Regler können ohne gegenseitigen Informationsaustausch ihre jeweiligen Regelgrößen stellen.

Eine verteilte Regelung bietet aber auch die Möglichkeit zur Realisierung einer prädiktiven Regelung. Dabei wird das Systemverhalten von einem Prädiktor vorhergesagt, um auf eine Veränderung in der Regelstrecke zu reagieren, bevor sie gemessen werden kann.

Bei einer Klimaregelung kann die relative Luftfeuchte sehr gut vorhergesagt werden. Zum Einen durch den Temperaturverlauf, von dem die Luftfeuchte direkt abhängt. Zum Anderen kann die in das System gebrachte Feuchtigkeit sehr gut bestimmt werden, wie zum Beispiel in einem Museum, wo an einem regnerischen Tag bei jedem neuen Besucherstrom Feuchtigkeit in die Räumlichkeiten gelangt.

4.1.2 Hybrider Regler

Es werden grundsätzlich zwei Gruppen von Reglern unterschieden:

Der Analogregler hat eine unendlich große Auflösung und wird meist mittels Operationsverstärkern realisiert. Ein analoger Input kann z.B. ein pt100[9] Temperaturfühler sein.

Beim Digitalregler wird die Eingangsgröße zeitlich abgetastet und quantisiert. Zusätzlich wird ein Halteglied verwendet, welches dem Regler für die gesamte Abtastzeit einen konstanten Wert bereitstellt. Für jeden Zeitabschnitt berechnet der Regler aus der Eingangsgröße eine entsprechende Ausgangsgröße.

Von einer hybriden Regelung spricht man, wenn die Regelung zeitkontinuierliche (analoge) und zeitdiskrete (digitale) Komponenten enthält.

[9] Temperatursensor aus Platin, beruhend auf der Widerstandsänderung unter Temperatureinfluss.

4.1.3 Zeitverhalten in SciCos

SciCos bietet Simulationsblöcke für zeitkontinuierliches und zeitdiskretes Modellverhalten. Somit lassen sich Analog-, Digital- sowie Hybridregler modellieren. Will man eine realistische Regelstrecke oder zusätzlich ein Umgebungsmodell simulieren, so sind kontinuierliche Blöcke im Projekt zwingend erforderlich.

Die kontinuierlichen Blöcke in SciCos sind immer aktiv bzw. vererben ihre Aktivierung (siehe Kapitel 3.1.1), die zeitdiskreten werden über einen Takteingang getaktet.

4.1.4 Zeitverhalten in OMNeT++

Nach der Code Generation gibt es kein wirkliches kontinuierliches Verhalten mehr. Die kontinuierlichen sowie die zeitdiskreten SciCos-Blöcke werden in OMNeT++ gleich behandelt. Sie werden nur durch ein entsprechendes Event aufgerufen. Dieses Event kann von einem anderen Block durch eine Message ausgelöst werden, oder durch eine „self-Message“ vom Block selbst kreiert werden. Der erste Fall entspräche einer Vererbung der Aktivierung vom Vorgängerblock, der zweite Fall einer Aktivierung über den Takteingang.

So kann ein unterschiedlichstes Verhalten der einzelnen Blöcke simuliert werden:

- Ein quasi-kontinuierliches Verhalten (z.B. das der Regelstrecke) kann durch einen schnellen Aufrufzyklus simuliert werden.
- Zum Beispiel kann bei einem Sensor eine langsamere Abtastrate verwendet werden, je nachdem wie oft die beobachtete Größe gemessen werden muss.
- Es kann ein zuvor in SciCos kontinuierlicher Block, nun vollkommen neu und auch unabhängig von anderen Blöcken, getaktet werden.

Dadurch können auch IEEE 802.15.4-Knoten simulieren, die eine begrenzte oder unzureichende Stromversorgung besitzen, wie es zum Beispiel bei einem Harvester-Knoten der Fall ist. Für das Senden einer Nachricht per Funk wird im Vergleich zum Energieverbrauch eines Mikrocontrollers viel Energie benötigt. So kann ein Knoten einen Wert

beinahe kontinuierlich messen, jedoch nicht ständig zur Weiterverarbeitung an andere Knoten senden. Dieses Verhalten kann durch ein Anpassen des Taktes eines Netzknotens realisiert werden.

Ein weiterer Vorteil des hybriden Modells von SciCos ist, dass bekannte und gebräuchliche Entwurfsverfahren für kontinuierliche Regler genutzt werden können, die unter der Einhaltung der Abtastrate auch in OMNeT++ als Digitalregler simulierbar sind. In der Theorie reicht eine Abtastrate nach Nyquist-Shannon mit:

$$\omega_{abtast} > 2\ \omega_{max}$$

In [23] wird jedoch für die Praxis ein Wert von

$$\omega_{abtast} > 6...20\ \omega_{max}$$

empfohlen. So behält der Regler ein gutes Führungs- und Störgrößenverhalten.

Nach der Code-Generierung sind die Beschränkungen in der Blockaktivierung von SciCos nicht mehr vorhanden. Es kann jede beliebige Aufruf-Situation simuliert werden. Dabei kann man sich an das SciCos-Modell halten oder bei Bedarf auch andere Takte verwenden. Dadurch ergeben sich viele neue Möglichkeiten, es kann bei falscher Anwendung aber auch zu Fehlern oder zu komplexem und unvorhersehbarem Verhalten führen.

4.1.5 IEEE 802.15.4-Netzwerk

IEEE 802.15.4 ist ein Funknetzstandard, der Kurzstreckenfunk zwischen intelligenten Geräten ermöglicht. Das Hauptaugenmerk liegt dabei, im Vergleich zu WLAN-Modulen, auf einem geringen Energieverbrauch. So soll ein IEEE 802.15.4-Modul nur mit Batterien ausgerüstet, mehrere Jahre wartungsfrei betrieben werden können. Außerdem sind die Module klein sowie leicht zu installieren und eignen sich so für die Haus- und Gebäudeautomation.

Es werden drei unterschiedliche Netztopologien unterstützt. So gibt es in der Stern- und Baum-Topologie hierarchische Beziehungen und einen Koordinator. In der Dritten, der vermaschten Form gibt es jedoch keine Beziehungen dieser Art. Hier sind alle Knoten gleichberechtigt, die gesamte Regelung wäre in diesem Fall ein Ad-hoc-Netzwerk.

Im Gegensatz zu den Simulationen in Kapitel 2, bei denen meist nur ein Teil der Regelung per Netzwerk überbrückt wird, ist hier die gesamte dezentrale Regelung ein drahtloses Netzwerk. Dadurch wirkt die Netzwerkkomponente nicht nur als ein einfaches variables Totzeitglied innerhalb einer Regelung, sondern es müssen alle in Kapitel 2.1 aufgeführten Netzeffekte beachtet werden, was eine vollständige Netzwerksimulation nötig macht.

Übertragungszeiten im IEEE 802.15.4-Netzwerk

Die Gesamtübertragungszeit zwischen zwei Netzwerkknoten setzt sich aus den folgenden Teilkomponenten zusammen:

- Latenzzeit
- Übertragungszeit (Datenrate, Paketgröße)
- Ausbreitungsgeschwindigkeit
- Verarbeitungszeit des Protokollstacks

Die Latenzzeit kann laut [4] unter bestimmten Umständen bis zu 16 ms (keine Kollisionen, Interferenzen oder Störstrahlungen) garantiert werden. Die Latenzzeit kann aber auch auf wenige Millisekunden reduziert werden, je nachdem wie oft beim CSMA-Verfahren (Carrier Sense Multiple Access) der Status des Übertragungskanals geprüft wird. Ein häufiges Prüfen und somit eine kurze Latenzzeit verkürzt jedoch die Batterielaufzeit eines Moduls.

$$t_l = 3...16\ ms$$

Die Übertragungszeit eines Paketes ergibt sich aus Paketgröße und Datenrate. Die Paketgröße setzt sich aus Nutzdaten und Overhead zusammen.

Den Overhead beeinflusst die gewählte Adressierungsart. Es gibt eine kurze (16-bit) und eine erweiterte (64-bit) Adressierung. Somit kann die Paketgröße zwischen 15 Byte (keine Nutzdaten) und 133 Byte (maximale Nutzdaten) schwanken.

Laut [24] ermöglicht der IEEE 802.15.4-Standard drei verschiedene Datenraten:

- 20 kbps bei 868 MHz
- 40 kbps bei 915 MHz
- 250 kbps bei 2,4 GHz

Daraus ergibt sich eine minimale und eine maximale Übertragungszeit:

$$t_{min} = 15\ Byte/250\ kbps = 0,48\ ms$$

$$t_{max} = 133\ Byte/20\ kbps = 53,2\ ms$$

Die Ausbreitungsgeschwindigkeit erfolgt in Lichtgeschwindigkeit und liegt bei kurzen Entfernungen im Nanosekunden-Bereich und kann somit vernachlässigt werden.

Die Verarbeitungszeit des Protokollstacks muss vernachlässigt werden, da hierüber keine gesicherte Aussage getroffen werden kann. Ein aktueller IEEE 802.15.4-Controller von Atmel hat einen CPU-Takt von 4 MHz. Das sagt jedoch wenig über die tatsächliche Verarbeitungsgeschwindigkeit aus. Sie liegt sicherlich im Bereich von Mikrosekunden. Eine genaue zeitliche Bestimmung ist aber nur schwer möglich und wird auch in den Simulationen aus Kapitel 2 nicht durchgeführt.

So erhält man, ohne Beachtung der Verarbeitungszeit durch den Controller, eine mögliche Zeitspanne von $t_G = 3,5...69\ ms$. Auch wenn diese Zeitspanne ziemlich groß erscheint, ist die wirkliche Variable in einer verteilten Regelung über ein Drahtlos-Netzwerk, nicht die bloße Übertragungszeit zwischen zwei Knoten. Die Übertragungszeit über mehrere Knoten hinweg, also über mehrere Hops[10], kann viel schwieriger kalkuliert werden. Bei jedem Hop wird die komplette Übertragungszeit benötigt, so dass eine Übertragung über zehn Hops die zehnfache Zeit in Anspruch nehmen würde. Bei 15

[10] Der Weg von einem Knoten zum nächsten, innerhalb eines Netzwerkes.

Hops kann es schon zu einer maximalen Verzögerung von einer Sekunde kommen. So kann die theoretische Übertragungszeit $t_G = 3,5...1\ s$ betragen.

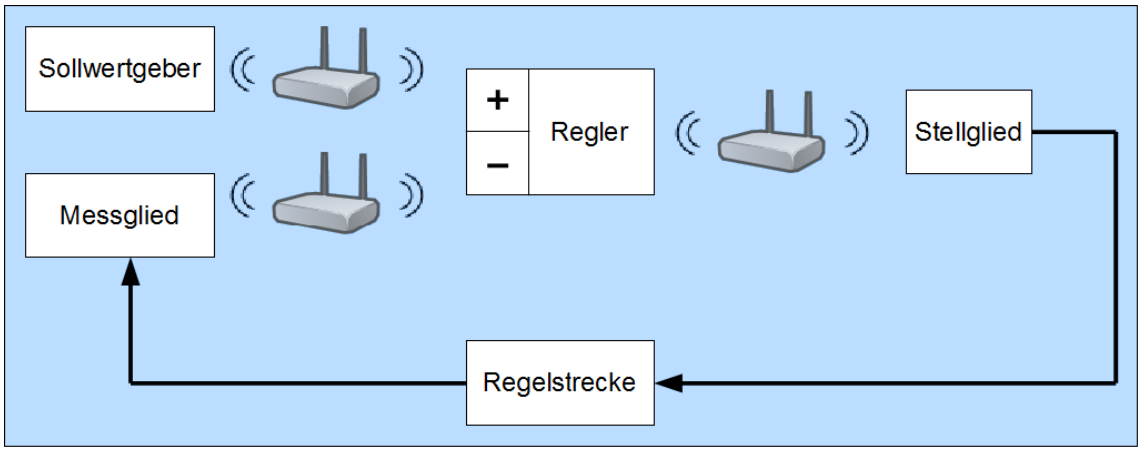


Abbildung 4.1: Regelkreis mit drahtlosem Regelmodul

Abbildung 4.1 zeigt, dass ein vollständiger Regelkreis normalerweise zwei Funkstrecken, die von Messglied zu Regler und die von Regler zu Stellglied besitzt. Daraus ergibt sich wiederum eine Verdopplung der Gesamttotzeit im Regelkreis. Da es aber dem Entwickler vollkommen frei steht, welche Komponenten er auf einem Netzwerkknoten zusammenfasst, kann dies vermieden werden (z.B. wenn Regler und Stellglied auf einem Knoten liegen). Das bedeutet aber auch, dass eine umfangreichere Regelung noch mehr drahtlose Verbindungen erfordern kann, wobei jeweils eine zusätzliche Totzeit in der Regelung auftritt.

Problematisch wird es, wenn Knoten ausfallen oder sie ihre Verbindung zu benachbarten Knoten verlieren. Dies würde ein Neu-Routing des gesamten Netzwerkes erforderlich machen, was zeitlich schwierig abzuschätzen ist, aber mit einigen Sekunden gerechnet werden muss. Eine Regelung kann also auf die üblicherweise auftretende Totzeit t_G hin optimiert werden. Es muss aber auch eine maximale Ausfallzeit durch den Ausfall von Knoten mit einhergehendem Neu-Routing beachtet werden, in der die Regelung trotzdem stabil bleiben muss.

Anwendungsfälle

Aus diesen Überlegungen kann abgeleitet werden, dass eine Regelung einer Raumbeleuchtung problematisch werden kann. Will man zum Beispiel die Raumbeleuchtung an

die augenblickliche Sonneneinstrahlung anpassen, so muss das Stellglied ohne Verzögerung auf eine Änderung der Sonneneinstrahlung, beispielsweise hervorgerufen durch vorbeiziehende Wolken, reagieren. Wenn jedoch Totzeiten von mehreren Sekunden im Regelkreis auftreten, so ist dies nicht möglich. Hier müssten die kritischen Verbindungen fest verdrahtet sein oder Sensor und Aktor zusammen auf einem Knoten liegen.

Vereinfacht kann gesagt werden, dass alle Regelungen denkbar sind, bei denen Totzeiten im Sekundenbereich nicht zur Instabilität führen. Dazu gehört in der Gebäudeautomatisierung:

- Temperaturregelung
- Luftfeuchtigkeitsregelung

Im Heimautomatisierungsbereich überwiegen Steuerungen, wie zum Beispiel für die Rollladen, die Beleuchtung oder die Fenster. Eine Regelung ist hier in den seltensten Fällen notwendig.

In Industrieanwendungen haben Regelungen aber eine viel größere Bedeutung. Hier gibt es eine Vielzahl von denkbaren Anwendungsfällen:

- Positionsregelungen
- Füllstandsregelungen
- Durchflussregelungen
- Geschwindigkeitsregelungen
- Klimaregelungen

4.2 Modellierung einer Regelungsanwendung mit SciCos

Beim Erstellen der Regelungsanwendung steht die Realisierbarkeit in einem Netzwerk im Vordergrund. Abhängig vom späteren Einsatzort der Anwendung wird die Regelung entworfen. So muss bedacht werden, welche Blöcke später auf einem Netzwerkknoten

eine bestimmte Anwendung ergeben. Diese Blöcke werden In SciCos zu einem Super-Block zusammengefasst.

In dem Beispiel aus Kapitel 3.1.1, Abbildung 3.1 ist der PID-Regler in einen Super-Block integriert worden. Dieser würde demnach einem Netzwerkknoten entsprechen (Abbildung 4.2). Es wäre sinnvoll, den Summationsblock vor dem Regler in den Super-Block zu integrieren. So entstünde ein Knoten, der Soll- und Istwert als Eingangsgrößen hat und eine Regelgröße ausgibt. Ein SciCos-Block außerhalb eines Super-Blocks kann später nicht Teil des Netzwerkes und somit kein Teil der Anwendung sein.

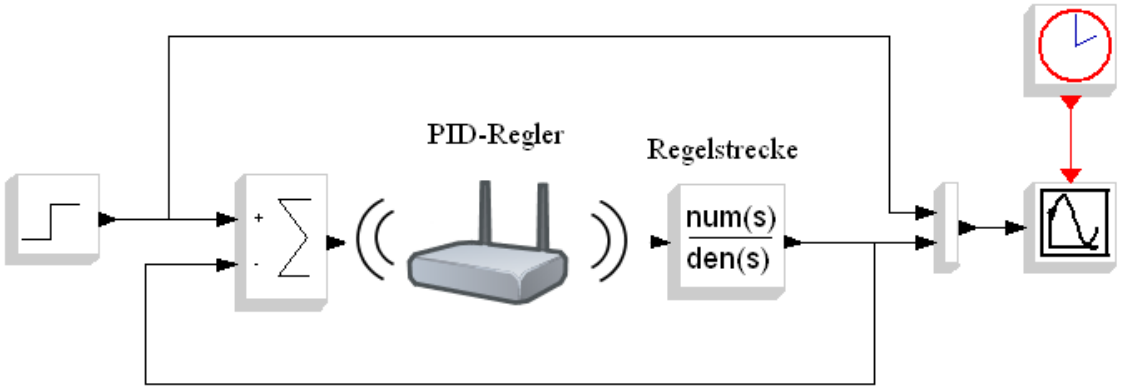


Abbildung 4.2: PID-Regler als Netzknoten

Es muss immer bedacht werden, wie eine Anwendung aufgebaut werden soll, was verteilt wird und was nicht verteilt werden darf. Der Vorteil ist, dass durch SciCos keinerlei Begrenzungen für die Regelungsanwendung entstehen. Jeder beliebige Verteilungsgrad kann modelliert werden, so wie es das spätere Umfeld der Anwendung erfordert. Dies ist ein großer Vorteil im Gegensatz zu den meisten anderen Co-Simulationen, bei denen oft nur ein Teil der Regelung mit einem Netzwerk überbrückt wird (Kapitel 2.5) oder nur bestimmte Layouts möglich sind (Kapitel 2.2).

Auch wenn jede denkbare Regelung möglich ist, gibt es Einschränkungen bei der Umsetzung, die zu beachten sind. Schließlich ist die Grundlage für eine Simulation in OMNeT++, dass die verwendeten SciCos-Blöcke in aktuellem C-Code vorliegen. So gibt es Blöcke, die nur in FORTRAN-Code vorhanden sind, andere liegen zwar in C-Code vor, sind aber veraltet. Sie können aber mit einigen Anpassungen trotzdem verwendet werden. Es wurde lediglich ein neuer Header mit einer neuen Blockstruktur eingeführt,

bei dem sich der Datentyp von einigen Variablen geändert hat.

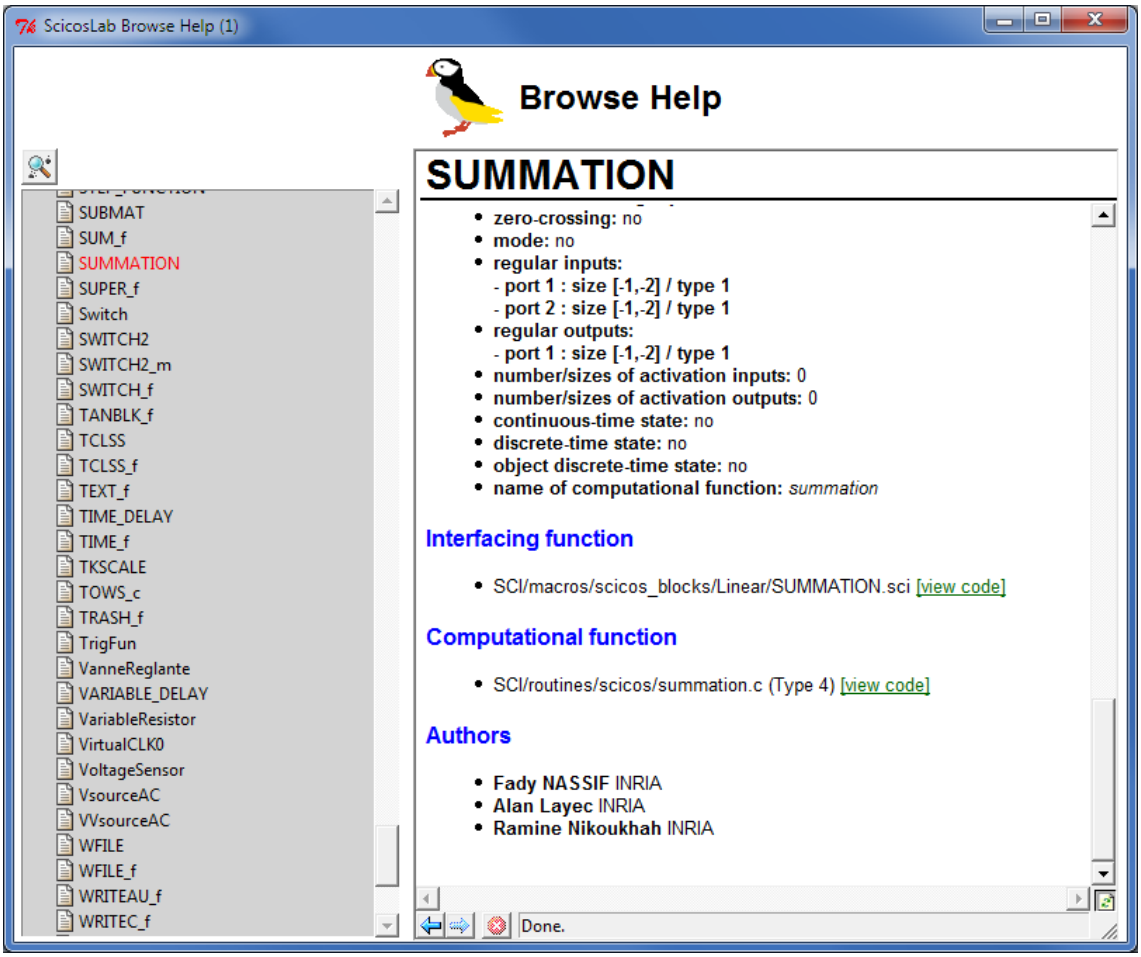


Abbildung 4.3: SciCos-Hilfe zum Summations-Block

Um solche Anpassungen zu vermeiden, sollten die Blöcke aus der Palette „OldBlocks“ nicht verwendet werden, da diese ausschließlich in veraltetem Code vorliegen. Allerdings kann es auch vorkommen, dass vermeintlich aktuelle Blöcke keinen aktuellen C-Code hinterlegt haben. Überprüft werden kann dies, indem die Hilfe des entsprechenden Blocks geöffnet wird (Abbildung 4.3). An dieser Stelle ist dokumentiert, in welchem „Type“ die „computational function“, also die Funktion in C-Code, vorliegt. Type 4 kennzeichnet den aktuellen C-Code, wodurch der Block problemlos verwendet werden kann. Unter „view code“ kann die Funktion eingesehen werden. Als Beispiel ist der Code des Summations-Block in Listing 4.1 aufgeführt:

```
#include "math.h"
#include "scicos_block4.h"
void summation(scicos_block *block,int flag){
  int j,k;
  double *u;
  int nu,mu;
  double *y;
  int *ipar;
  y=GetRealOutPortPtrs(block,1);
  nu=GetInPortRows(block,1);
  mu=GetInPortCols(block,1);
  ipar=GetIparPtrs(block);
  if(flag==1){
    if (GetNin(block)==1){
      y[0]=0.0;
      u=GetRealInPortPtrs(block,1);
      for (j=0;j<nu*mu;j++) {
        y[0]=y[0]+u[j];
      }
    }
    else {
      for (j=0;j<nu*mu;j++) {
        y[j]=0.0;
        for (k=0;k<GetNin(block);k++) {
          u=GetRealInPortPtrs(block,k+1);
          if(ipar[k]>0){
            y[j]=y[j]+u[j];
          }else{
            y[j]=y[j]-u[j];
          }
        }
      }
    }
  }
}
```

Listing 4.1: Summation

Im Gegensatz dazu ist in Listing 4.2 ein veralteter Block in FORTRAN abgebildet:

```
1  subroutine minblk(flag,nevprt,t,xd,x,nx,z,nz,tvec,ntvec,
2       &     rpar,nrpar,ipar,nipar,u,nu,y,ny)
3  c     Copyright INRIA
5  c     Scicos block simulator
6  c     outputs the minimum of all inputs
7  cc
8        double precision t,xd(*),x(*),z(*),tvec(*),rpar(*),u(*),y(*)
9        integer flag,nevprt,nx,nz,ntvec,nrpar,ipar(*)
10       integer nipar,nu,ny
11 c
12       double precision ww
13 c
14       ww=u(1)
15       do 15 i=1,nu
16          ww=min(ww,u(i))
17  15   continue
18       y(1)=ww
20       end
```

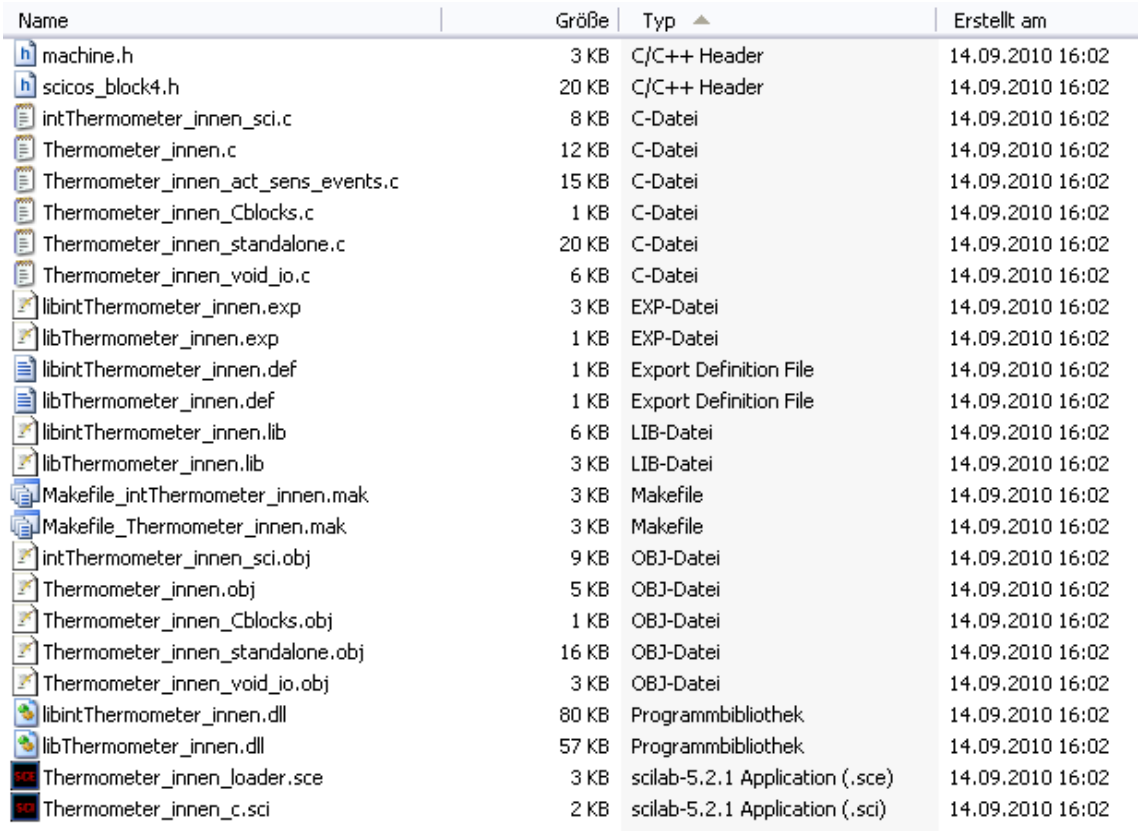
Listing 4.2: Minimum

Bei der Code-Generierung stellen die Taktleitungen ein großes Problem dar. Aktivierende Outputs sind innerhalb eines Super-Blocks nicht zulässig (Vgl. Kapitel 12.2.2 aus [20]), da sich ein C-Code-Block nicht selbst aktivieren kann. Das bedeutet, dass keine Clock-Blöcke innerhalb eines Super-Blocks erlaubt sind. Solche Aktivierungen sind als Takteingang von außen an den Super-Block anzulegen, wodurch sie vom Code-Generator ignoriert werden. Im Quellcode wird ein Standard-Takt verwendet, der nachträglich geändert werden kann.

Es muss jedoch nicht auf alle Blöcke aus der „Events-Palette“ verzichtet werden. Der `If-Then-Else`- und der `Event-Select`-Block sind zulässig. Wobei der `If-Then-Else`-Block unerlässlich ist. Mit ihm ist es möglich, Bedingungen zu formulieren, die für einen intelligenten Netzknoten zwingend notwendig sind.

4.3 Der generierte Quellcode

Nach erfolgreicher Code-Generierung werden die generierten Dateien (Abbildung 4.4) im zuvor ausgewählten Verzeichnis abgelegt. SciCos greift nun beim Simulieren direkt auf diese Dateien zu. Sie dürfen deshalb nicht entfernt oder verschoben werden.



Name	Größe	Typ	Erstellt am
machine.h	3 KB	C/C++ Header	14.09.2010 16:02
scicos_block4.h	20 KB	C/C++ Header	14.09.2010 16:02
intThermometer_innen_sci.c	8 KB	C-Datei	14.09.2010 16:02
Thermometer_innen.c	12 KB	C-Datei	14.09.2010 16:02
Thermometer_innen_act_sens_events.c	15 KB	C-Datei	14.09.2010 16:02
Thermometer_innen_Cblocks.c	1 KB	C-Datei	14.09.2010 16:02
Thermometer_innen_standalone.c	20 KB	C-Datei	14.09.2010 16:02
Thermometer_innen_void_io.c	6 KB	C-Datei	14.09.2010 16:02
libintThermometer_innen.exp	3 KB	EXP-Datei	14.09.2010 16:02
libThermometer_innen.exp	1 KB	EXP-Datei	14.09.2010 16:02
libintThermometer_innen.def	1 KB	Export Definition File	14.09.2010 16:02
libThermometer_innen.def	1 KB	Export Definition File	14.09.2010 16:02
libintThermometer_innen.lib	6 KB	LIB-Datei	14.09.2010 16:02
libThermometer_innen.lib	3 KB	LIB-Datei	14.09.2010 16:02
Makefile_intThermometer_innen.mak	3 KB	Makefile	14.09.2010 16:02
Makefile_Thermometer_innen.mak	3 KB	Makefile	14.09.2010 16:02
intThermometer_innen_sci.obj	9 KB	OBJ-Datei	14.09.2010 16:02
Thermometer_innen.obj	5 KB	OBJ-Datei	14.09.2010 16:02
Thermometer_innen_Cblocks.obj	1 KB	OBJ-Datei	14.09.2010 16:02
Thermometer_innen_standalone.obj	16 KB	OBJ-Datei	14.09.2010 16:02
Thermometer_innen_void_io.obj	3 KB	OBJ-Datei	14.09.2010 16:02
libintThermometer_innen.dll	80 KB	Programmbibliothek	14.09.2010 16:02
libThermometer_innen.dll	57 KB	Programmbibliothek	14.09.2010 16:02
Thermometer_innen_loader.sce	3 KB	scilab-5.2.1 Application (.sce)	14.09.2010 16:02
Thermometer_innen_c.sci	2 KB	scilab-5.2.1 Application (.sci)	14.09.2010 16:02

Abbildung 4.4: Vom Code-Generator erzeugte Dateien

Um den Standalone-Code zu testen, kann dieser zu einer ausführbaren Datei kompiliert und gelinkt werden. Dabei handelt es sich um ein Konsolenprogramm, welches die Eingänge vom Benutzer per Tastaturabfrage einliest, um daraus die Ausgänge zu berechnen. Im folgenden Abschnitt der Arbeit wird das Vorgehen bei der Erzeugung einer *standalone.exe*-Datei beschrieben.

Die Datei *Blockname.c* ist die „computational function“, die direkt von SciCos aufgerufen wird. Die Standalone-Datei ist mit *Blockname_standalone.c* benannt. Sie enthält die „computational function“ des Standalone-Codes. Insgesamt werden die in Abbildung 4.5 dargestellten Dateien zum Kompilieren der *standalone.exe* benötigt.

Abbildung 4.5: Für eine Standalone-Datei benötigte Quellen

In der Datei *Blockname_act_sens_events.c* sind die Funktionen zum Lesen und Schreiben in der Konsole implementiert. Die Datei *Blockname_Cblocks.c* beinhaltet nur eine leere Funktion, sie wird aber trotzdem zum Erstellen benötigt.

Sind diese Dateien vorhanden, kann in einer Kommandozeile der Quellcode kompiliert und gelinkt werden. Der Konsolenbefehl zum Erstellen der EXE-Datei ist in Abbildung 4.6 dargestellt. Das Makefile *Makefile_Name.mak* wird eingelesen und die darin angegebenen Schritte zum Erstellen der Datei werden ausgeführt. Auf dem Computer muss dazu „nmake“ installiert sein. Dafür wird ebenfalls ein Compiler und ein Linker benötigt. Auf dem Testrechner wurde dafür Visual Studio[11] genutzt. Dem Compiler müssen ebenfalls die Windows- und SciCos-Standard-Include-Dateien bekannt sein. Dafür müssen die entsprechenden Header-Dateien in den Ordner mit dem Quellcode kopiert werden. Es können aber auch im Makefile die Pfade der benötigten Dateien angegeben werden.

Abbildung 4.6: Befehl „nmake“ zum Erstellen der Standalone-Datei

[11] Entwicklungsumgebung von Microsoft für verschiedene Hochsprachen.

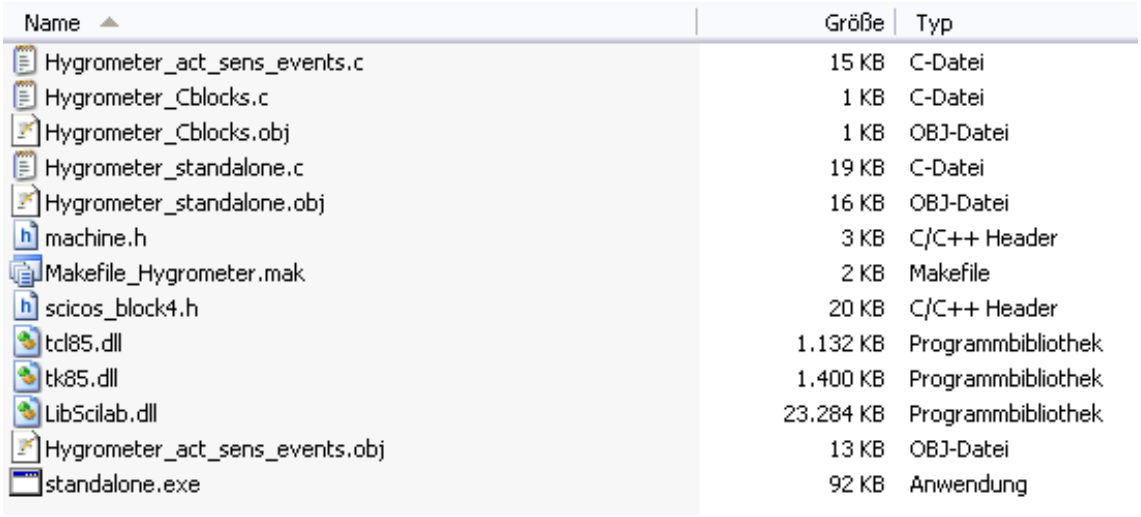


Abbildung 4.7: Ordner mit erstellter Standalone-Datei und SciCos-DLLs

Wenn das Kompilieren und Linken fehlerfrei funktioniert hat, ist die *standalone.exe* und eine Objekt-Datei erstellt worden. Zum Ausführen der EXE-Datei werden zusätzlich drei SciCos-DLLs (Dynamic Link Library) benötigt, die sich im gleichen Ordner wie die EXE-Datei befinden müssen (Abbildung 4.7). Nun kann die Standalone-Datei gestartet werden. In der sich öffnenden Konsole werden Simulationsparameter ausgegeben und die Eingänge des Blocks werden vom Nutzer abgefragt (Abbildung 4.8). Anhand der Ausgabe und dem aus SciCos bekannten Verhalten, kann man erkennen, ob der Standalone-Code das erwartete Verhalten zeigt.

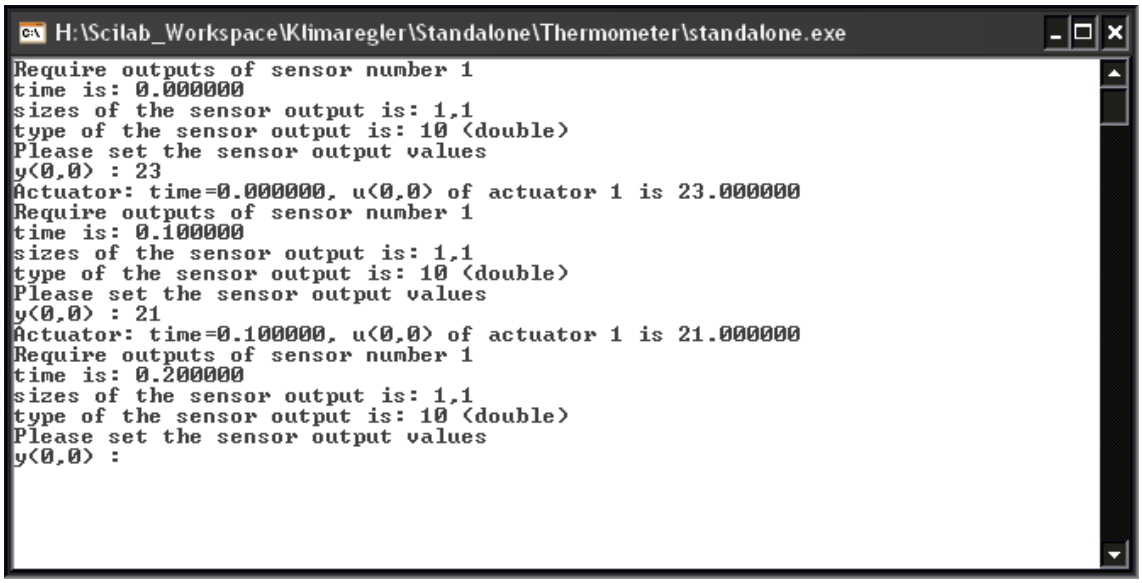


Abbildung 4.8: Standalone.exe

4.4 Simulation mit OMNeT++

4.4.1 Integration als C- oder C++-Code?

Der von SciCos generierte Code ist C-Code und kann als „extern C“ in die Entwicklungsmgebung von OMNeT++ eingebunden werden. Bei mehreren Blöcken gibt es aber Probleme mit den globalen Variablen des C-Codes. Da es auch für das Fraunhofer-Projekt unhandlich ist mit C-Code zu arbeiten, wurde die direkte Einbindung des C-Codes verworfen.

Für jeden Block wurde deshalb eine eigene C++-Klasse beziehungsweise ein eigenes OMNeT++-Modul erstellt, in das der C-Code integriert wurde. Dabei müssen aber einige Dinge beachtet werden, da der C-Code ohne Anpassungen in einer C++-Umgebung nicht lauffähig ist:

- Die Funktionsköpfe des generierten Codes sind in einer speziellen C-Deklaration angelegt, dieser ist aber nicht mit der C++-Schreibweise kompatibel. Die Übergabeparameter müssen komplett in der Klammer deklariert werden, was in C aufgeteilt werden kann (Listing 4.3).

```
1 int Thermometer_sim(tf, dt, h, solver, typin, inptr, typout, outptr)
2    double tf, dt, h;
3    int solver, *typin, *typout;
4    void **inptr, **outptr;
5 {...}
```

Listing 4.3: C-Funktionskopf

- Wie in Listing 4.4 zu sehen ist, muss in C++ der Klassenname vor der Funktion stehen. Bei Blöcken mit vielen Funktionen zieht dies einen großen Aufwand nach sich.

```
1 int Thermometer::Thermometer_sim( double tf, double dt, double h,
      int solver, int *typin, void **inptr, int *typout, void **outptr)
2 {...}
```

Listing 4.4: C++ Funktionskopf

- In C kann einem double-Pointer ein void-Pointer zugewiesen werden (Listing 4.5). In C++ ist dies aber nicht erlaubt, so dass an dieser Stelle ein Typecast, entsprechend Listing 4.6 erfolgen muss.

```
1 void **_work = GetPtrWorkPtrs(block);
2 double *rw;
3 rw = *_work;
```

Listing 4.5: Void-Pointer-Initialisierung in C

```
1 void **_work = GetPtrWorkPtrs(block);
2 double *rw;
3 rw = ( double* )*_work;
```

Listing 4.6: Void-Pointer-Initialisierung in C++ mit Typecast

4.4.2 Code-Anpassungen in OMNeT++

Benötigte Dateien

Die Anpassungen des Quellcodes, die über die reinen Sprachanpassungen hinausgehen, sind umfangreicher. Bevor der konkrete Quellcode betrachtet werden kann, müssen die erforderlichen Dateien bekannt sein. Bei der Code-Generierung werden über 20 Dateien erzeugt (Abbildung 4.4). Für das Erstellen einer EXE-Datei sind davon die Dateien aus Abbildung 4.5 erforderlich, von denen für den Import in OMNeT++ lediglich die Quellcode-Dateien

- *Name_standalone.c*
- *Name_act_sens_events.c*

und die Header

- *scicos_block4.h*
- *machine.h*

benötigt werden.

Benötigte Funktionen

Die in Kapitel 4.3 betrachteten Standalone-Dateien sind zum Erstellen einer fertigen Anwendung vorgesehen. Deshalb sind darin einige Funktionen implementiert, die für das Blockverhalten nicht zwingend notwendig sind. Dazu gehört eine Auswertung des Übergabeparameters an die EXE-Datei sowie eine Fehlerbehandlung. Bei der manuellen Implementierung wird noch keine Fehlerbehandlung notwendig sein, sondern erst bei einer eventuell automatisierten Erstellung des OMNeT++-Modells.

Durch die Vielzahl an Funktionen ist es einfacher, lediglich die zu betrachten, die erforderlich sind, und diese in das OMNeT++ Modell zu integrieren. Dies sind die Funktionen:

- *Name_sim (...)*
- *Name_actuator (...)*
- *Name_sensor (...)*
- „computational functions“ der verwendeten SciCos-Blöcke

Die *Name_sim*-Funktion wird bei Verwendung als EXE-Datei von der C-Main-Datei aufgerufen. Diese Funktion muss nun die entsprechende OMNeT++-Klasse, die für jeden SciCos-Super-Block erstellt wird, übernehmen. Beim Aufruf der Funktion wird ein Pointer für Ein- und Ausgang, die Dimension der Eingangs- und Ausgangsports sowie die Variablen

- *tf* - final time (Ende-Zeit)
- *dt* - clock time (Takt-Zeit)
- *h* - solver step (Solver-Schrittweite)
- *s* - type of solver (Solver-Typ: Heun-, Euler- oder Runge-Kutta-Verfahren)

übergeben.

Struktur der Simulationsfunktion

Der Aufbau der Simulationsfunktion muss im Gegensatz zur EXE-Datei leicht abge-

wandelt werden. Die Funktion läuft normalerweise nach ihrem Aufruf bis zur Ende-Zeit *tf* in einer while-Schleife und liefert dabei in jedem Takt *dt* einen Ausgangswert. Im OMNeT++-Modell soll aber bei einem Aufruf des Super-Blocks lediglich ein Ausgangswert berechnet und danach die Simulationsfunktion wieder verlassen werden. Dazu wurde die while-Schleife durch eine if-Anweisung ersetzt.

Listing 4.7 zeigt den Aufbau des generierten Codes. Im Gegensatz dazu ist der für OMNeT++ angepasste Code in Listing 4.8 abgebildet.

```
1  void Name_sim(...)
2  {
3    /* Variablendefinition */
4    ...
5    /* Initialisierungsaufruf der Blöcke*/
6    ...
7    while(t<=tf)
8    {
9      /* Rechenaufruf der Blöcke */
10   }
11 }
```

Listing 4.7: Ursprüngliche Struktur der Simulationsfunktion

```
1  void Name_sim(...)
2  {
3    /* Variablendefinition */
4    ...
5    if (t==0)
6    {
7      /* Initialisierungsaufruf der Blöcke*/
8      ...
9    }
10   else if(t<=tf)
11   {
12     /* Rechenaufruf der Blöcke */
13   }
14 }
```

Listing 4.8: Geänderte Struktur der Simulationsfuntion

Aufruf der Blöcke

Innerhalb der Simulationsfunktion werden die einzelnen Blöcke des Super-Blocks sowie die Sensor- und Aktuator-Funktion aufgerufen. Dabei wird eine Struktur mit dem Blockvariablen und ein Flag übergeben. Die Blockvariablen legen die Eigenschaften des jeweiligen Blocks fest und werden im Abschnitt „Variablendefiniton“ definiert (Listing 4.9). Darin wird unter anderem die Größe des Ein- und Ausgangsportes festgelegt und die Ein- und Ausgangspointer übergeben. Das übergebene Flag bestimmt die auszuführende Aufgabe. So gibt es als Erstes immer einen Initialisierungsaufruf und danach einen Aufruf zur Berechnung des Ausgangswertes. Das Flag kann Werte von null bis neun annehmen, jedoch unterscheidet nicht jeder Block alle Flags.

```
1  /* set blk struc. of 'integral_func' (type 4 - blk nb 2) */
2  block_Regelstrecke[ 1 ].type = 4;
3  block_Regelstrecke[ 1 ].ztyp = 0;
4  block_Regelstrecke[ 1 ].ng = 0;
5  block_Regelstrecke[ 1 ].nz = 0;
6  block_Regelstrecke[ 1 ].nx = 1;
7  block_Regelstrecke[ 1 ].noz = 0;
8  block_Regelstrecke[ 1 ].nrpar = 0;
9  block_Regelstrecke[ 1 ].nopar = 0;
10 block_Regelstrecke[ 1 ].nipar = 0;
11 block_Regelstrecke[ 1 ].nin = 1;
12 block_Regelstrecke[ 1 ].nout = 1;
13 block_Regelstrecke[ 1 ].nevout = 0;
14 block_Regelstrecke[ 1 ].nmode = 0;
15 block_Regelstrecke[ 1 ].x = &( x[ 2 ] );
16 block_Regelstrecke[ 1 ].xd = &( xd[ 2 ] );
17 block_Regelstrecke[ 1 ].inptr = inptr_2;
18 block_Regelstrecke[ 1 ].insz = insz_2;
19 block_Regelstrecke[ 1 ].outptr = outptr_2;
20 block_Regelstrecke[ 1 ].outsz = outsz_2;
```

Listing 4.9: Initialisierung der Variablen einer Blockstruktur

Beim Aufruf der Sensor- und Aktuator-Funktion wird nicht die vollständige Funktion übergeben, sondern nur die notwendigen Werte aus der Struktur. Diese Aufrufart gleicht

der des veralteten Blockaufrufes, weswegen hier in späteren Versionen mit einer Anpassung zu rechnen ist.

Listing 4.10 zeigt eine Sequenz aus einer Simulationsfunktion. Die Reihenfolge beim Aufruf der Blöcke ist bei der Initialisierung unbedeutend, beim Rechenaufruf ist jedoch die Reihenfolge zu beachten. Zuerst werden die Eingänge mittels der Sensor-Funktionen (für jeden Eingang eine Funktion) eingelesen, danach werden die entsprechenden Blöcke des Super-Blocks aufgerufen. Abschließend schreibt die Aktuator-Funktion die Ausgangswerte. Da es sich um automatisch generierten Code handelt, muss jedoch die Reihenfolge nicht explizit geprüft oder angepasst werden.

```
1  ...
2  /* Call of 'capteur2' (type 0 - blk nb 9) */
3  block_PID_Regler[ 8 ].nevprt = 0;
4  local_flag = 1;
5  nport = 2;
6  PID_Regler_sensor( &local_flag, &nport, &block_PID_Regler[ 8 ].nevprt, &t,
       ( SCSREAL_COP * )block_PID_Regler[ 8 ].outptr[ 0 ], &block_PID_Regler
       [ 8 ].outsz[ 0 ], &block_PID_Regler[ 8 ].outsz[ 1 ], &block_PID_Regler
       [ 8 ].outsz[ 2 ], block_PID_Regler[ 8 ].insz[ 0 ], block_PID_Regler[ 8
        ].inptr[ 0 ] );

8  /* Call of 'summation' (type 4 - blk nb 8) */
9  block_PID_Regler[ 7 ].nevprt = 3;
10 local_flag = 1;
11 SciBlocks.summation( &block_PID_Regler[ 7 ], local_flag );

13 /* Call of 'gainblk' (type 4 - blk nb 1) */
14 block_PID_Regler[ 0 ].nevprt = 1;
15 local_flag = 1;
16 SciBlocks.gainblk( &block_PID_Regler[ 0 ], local_flag );
17 ...
18 /* Call of 'actionneur1' (type 0 - blk nb 7) */
19 block_PID_Regler[ 6 ].nevprt = 1;
20 local_flag = 1;
21 nport = 1;
22 PID_Regler_actuator( &local_flag, &nport, &block_PID_Regler[ 6 ].nevprt, &
```

```
t, ( SCSREAL_COP * )block_PID_Regler[ 6 ].inptr[ 0 ], &
block_PID_Regler[ 6 ].insz[ 0 ], &block_PID_Regler[ 6 ].insz[ 1 ], &
block_PID_Regler[ 6 ].insz[ 2 ], block_PID_Regler[ 6 ].outsz[ 0 ],
block_PID_Regler[ 6 ].outptr[ 0 ] );
```

Listing 4.10: Aufruf der Blöcke

Simulationsfunktionen der genutzten Blöcke

In der Simulationsfunktion werden lediglich die einzelnen Blöcke aufgerufen. Der Code der jeweiligen Simulationsfunktionen, wie zum Beispiel der Summationsblock in Listing 4.1, ist jedoch nicht im generierten Code zu finden. Der Grund dafür ist, dass die kompilierte *standalone.exe* dynamisch eine SciLab-DLL lädt, in der die Funktionen zu finden sind. Es ist allerdings nicht möglich eine nicht selbst erstellte DLL zu laden, weswegen hier ein anderer Weg eingeschlagen werden muss.

In Kapitel 4.2 wurde beschrieben, wie mittels der SciCos-Hilfe der Code der einzelnen Blöcke zu finden ist. Dieser Code kann somit auch problemlos in OMNeT++ integriert werden. Nachfolgend werden zwei Möglichkeiten des Imports verglichen:

1. Erstellen einer C++-Klasse:

   Der Code aller verwendeten Funktionen wird in eine eigene Klasse gekapselt. Muss die Simulationsfunktion eine solche Funktion eines Blocks aufrufen, benötigt sie den Header der Klasse, wodurch eine neue Instanz erstellt werden kann, mit der die gewünschte Funktion aufgerufen wird.

   Der Vorteil ist, dass der Code für die Funktionen zentral in einer einzigen Klasse gespeichert ist. Der Implementierungsaufwand ist bei dieser Realisierung jedoch ziemlich hoch, da in jedem Modul eine Instanz angelegt werden muss, dessen Name vor jedem Blockaufruf eingefügt werden muss.

2. Direktes Einbinden in die jeweilige Klasse:

   Die pro Super-Block genutzten Blockfunktionen werden in die jeweilige Klasse ko-

piert, wodurch sie innerhalb dieser Klasse ohne zusätzliche Deklarierungen aufgerufen werden können. Dadurch entfällt der Implementierungsaufwand der ersten Umsetzung, allerdings muss in jeder Klasse der Funktionscode neu integriert werden, was dazu führt, dass häufig genutzte Funktionen mehrfach im Quelltext vorkommen.

Actuator-Sensor-Funktionen

Beim Bilden eines Super-Blocks in SciCos werden für Ein- und Ausgangsports neue Blöcke hinzugefügt. Diese sind laut Definition normale SciCos-Blöcke und werden bei der Code-Generation auch als solche behandelt. Jedoch gibt es auch bei mehreren Ports nur jeweils eine Funktion für die Ein- und Ausgänge. Die Portnummer wird übergeben und dann per Switch-Case-Anweisung ausgewertet.

Der Input der Werte wird in der *Standalone.exe* als Tastaturabfrage mittels „scanf“ ausgeführt. In OMNeT++ muss diese Abfrage durch eine einfache Zuweisung ersetzt werden. Dazu wird im Header eine Input- und eine Output-Variable für jeden Port angelegt, welche die Ein- und Ausgangsgrößen aufnehmen. Diese Variablen erhalten ihre Werte vom jeweiligen Vorgänger-Block mittels OMNeT++-Message.

Die Actuator-Sensor-Funktionen können stark gekürzt werden. Der Datentyp der Ein- und Ausgangsgrößen wird in einer weiteren Switch-Case-Anweisung ausgewertet. Da aber pro Block nur ein Datentyp verwendet wird, können die anderen entfernt werden. Außerdem ist eine File-Ausgabe bzw. -Eingabe vorgesehen, die ebenso komplett entfernt werden kann.

Blocktypen

Es gibt zwei Arten von Blöcken:

Die konstanten, zeitunabhängigen Blöcke sind dadurch gekennzeichnet, dass bei einer konkreten Eingangsbelegung immer der gleichen Wert ausgeben wird. Der Aufrufzeitpunkt ist nicht von Bedeutung. Ein Beispiel für einen solchen Block ist der `Gain`-Block. Bei ihm kann ein Verstärkungsfaktor angegebenen werden, mit

dem der Eingang multipliziert wird. Am Ausgang wird immer das Ergebnis dieser Multiplikation ausgegeben. Für diese Berechnung wird kein Solver und somit auch keine der Solver-Variablen benötigt. Jedoch wird dies in der Code-Generierung von SciCos nicht berücksichtigt, weswegen diese Variablen auch hier vorhanden sind, aber nicht genutzt werden.

Zu dieser ersten Gruppe gehören die meisten Blöcke. Sie sind leicht umzusetzen, da hier der gesamte Rechenaufwand in der *Name_sim*-Funktion stattfindet. Somit sind keine zusätzlichen Funktionen notwendig.

Bei den kontinuierlichen, zeitabhängigen Blöcken ist der Aufrufzeitpunkt von Bedeutung. Es muss ein Differentialgleichungssystem gelöst werden, wozu der Solver benötigt wird und die Solver-Variablen „h“ und „s“ übergeben werden müssen. Am Beispiel des Integrals in Abbildung 4.9 wird dies deutlich. Das Integral über der Konstante $c = 1$ ergibt $y = x$, also eine Gerade durch den Punkt P(1,1). Wird der Integral-Block nun zum Zeitpunkt $t_1 = 1\ s$ aufgerufen so ist die Ausgangsgröße $y = 1$. Wird der Block aber zum Zeitpunkt $t_2 = 20\ s$ aufgerufen so liegt der Ausgangswert in diesem Beispiel bei $y = 20$.

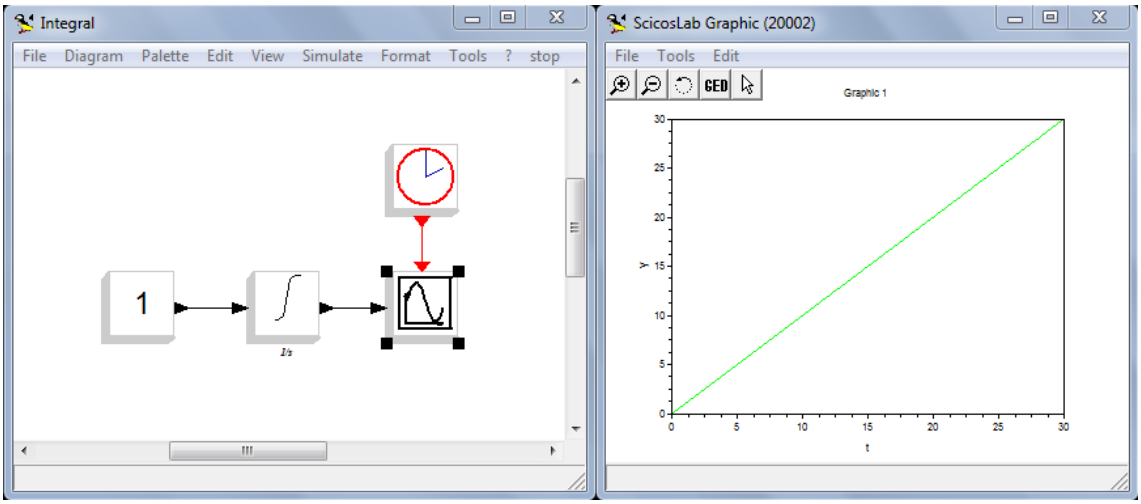


Abbildung 4.9: Integral über „1“

Zu der Gruppe der kontinuierlichen Blöcke gehören neben dem Integral-Block auch der Ableitungs- und der Übertragungsfunktions-Block, welche für das Modellieren einer Regelung sowie Regelungsumgebung von Bedeutung sind.

Diese Blöcke haben einen zeitdiskreten und einen zeitkontinuierlichen Quelltext-Abschnitt. Dazu wird eine weitere Simulationsfunktion, die *Name_simblk*-Funktion benötigt. Zusätzlich wird die Variable „phase“ eingeführt, die den augenblicklichen Zustand (zeitdiskret oder zeitkontinuierlich) angibt. Weiterhin ist der Code für die Solver-Funktionen implementiert. Als Beispiel ist der Code des Euler-Solvers in Listing 4.11 abgebildet. Die Bezeichnung „ODE“ (ordinary differential equation) steht für eine gewöhnliche Differentialgleichung.

```
1  /* Euler's Method */
2  int ode1( double *x, double *xd, double t, double h )
3  {
4    int i;
5    int ierr;
6    ierr = Regelstrecke_simblk( t, x, xd );
7    if( ierr != 0 )
8      return ierr;
9    for( i = 0; i < NEQ; i++ )
10   {
11     x[ i ] = x[ i ] + h * xd[ i ];
12   }
13   return 0;
14 }
```

Listing 4.11: Euler-Verfahren

Simulationsfunktion mit kontinuierlichen Blöcken

Enthält eine Simulationsfunktion eine oder mehrere kontinuierliche Blöcke, so sind weitere Anpassungen innerhalb der Simulationsfunktion erforderlich. Um zum Beispiel den Ausgangswert eines Integral-Blocks zu bestimmen, nutzt SciCos den Aufrufzeitpunkt des Blocks *t*, die Dauer der zu berechneten Zeitspanne *dt* und einen Startwert, beziehungsweise den Ausgangswert des letzten Aufrufs. Die Zeitwerte werden beim Aufruf der Simulationsfunktion übergeben, die Startwerte werden in der Initialisierungsphase des Blocks geladen (Listing 4.12) und bei jedem Durchlauf aktualisiert.

```
1 /* Continuous states declaration */
2 double x[] = { 0, 0, 50, 0, 0, 0, 0 };
3 double xd[] = { 0, 0, 0, 0, 0, 0, 0 };
```

Listing 4.12: Initialisierung der kontinuierlichen Blöcke

Bei der *standalone.exe* wird die while-Schleife nie verlassen, deshalb bleiben die Pointer, die diese Werte speichern, unangetastet. Da in OMNeT++ die Simulationsfunktion eines Blocks auch wieder verlassen wird, müssen diese Pointerwerte gespeichert und beim nächsten Aufruf des Blocks neu geladen werden. Dazu werden im Header der jeweiligen Klasse Hilfsvariablen angelegt. Bei mehr als einem kontinuierlichen Block innerhalb der Simulationsfunktion kann dies auch ein Array wie in Listing 4.13 sein.

```
1 /* def number of continuous state */
2 #define NEQ 7
3 double x_hilf[NEQ];
4 double xd_hilf[NEQ];
```

Listing 4.13: Anlegen der Hilfsvariablen

Bevor nun die Simulationfunktion verlassen wird, werden die Werte für jeden Block gespeichert (Listing 4.14). Beim nächsten Aufruf ($t! = 0$) werden diese Werte wieder geladen (Listing 4.15).

```
1 for( int i = 0; i <= NEQ; i++ )
2 {
3   x_hilf[ i ] = x[ i ];
4   xd_hilf[ i ] = xd[ 1 ];
5 }
```

Listing 4.14: Speichern der Werte

```
1 if( t != 0 )
2 {
3   for( int i = 0; i < NEQ; i++ )
4   {
5     x[ i ] = x_hilf[ i ];
```

```
6       xd[ 1 ] = xd_hilf[ i ];
7     }
8   }
```

Listing 4.15: Laden der Werte

Das Integral und die Übertragungsfunktion nutzen die Variablen *x* und *xd*, der Ableitungs-Block benutzt stattdessen einen „Workpointer“, welcher ebenfalls gespeichert werden muss.

4.4.3 Fertigstellung der Simulation

Sobald die einzelnen Simulationsfunktionen der Super-Blöcke korrekt arbeiten, kann die komplette OMNeT++-Simulation fertiggestellt werden. Für jeden Super-Block wird ein eigenes OMNeT++-Modul erstellt. Dieses Modul simuliert mit Hilfe der MiXiM- und SuSAN-Erweiterungen ein IEEE 802.15.4-Modul und stellt Funktionen für die Applikationsschicht bereit. Für eine Regelungsanwendung werden folgende Funktionen benötigt:

Initialize wird beim Start der Simulation aufgerufen. Die Initialisierung der Anwendung ist nicht zwingend notwendig, da aber der SciCos-Code ebenfalls einen Initialisierungsaufruf benötigt, kann diese Funktion dafür verwendet werden. Während der Initialisierung ist die an die Simulationsfunktion übergebene Simulationszeit $t = 0$. Dies wertet die Simulationsfunktion aus und führt die Initialisierungsaufrufe aus. In den Actuator-Sensor-Funktionen kann der Nutzer eigene Initialisierungen an diesen Schnittstellen hinzufügen. Zusätzlich können die eigenen Hilfsvariablen sowie die Ein- und Ausgangsvariablen, die im Header deklariert worden, an dieser Stelle initialisiert werden.

HandleMessage ist die Hauptfunktion für die Anwendung. Diese Funktion wird beim Eintreffen neuer Nachrichten aufgerufen. Hier erfolgt der Aufruf der Simulationsfunktion jedes einzelnen Blocks.

HandleSelfMsg wird aufgerufen, wenn ein Modul sich selbst eine Nachricht schickt.

Messages senden

Die Module können sich gegenseitig (HandleMessage) und selbst (HandleSelfMsg) Messages senden. Dazu müssen sie ihre eigene ID und die ID des Zielmoduls kennen. Die Ermittlung der ID's kann vor jedem Senden oder einmalig im Initialisierungsprozess erfolgen. Zum Verschicken von Messages muss ein eigenes Paket definiert werden, welches die zu übermittelnden Werte, sowie Ziel- und Quelladresse aufnehmen kann.

In Listing 4.16 werden einem Paket die benötigten Werte übergeben, um es anschließend an das Zielmodul zu senden.

```
1 /* Paket anlegen */
2 pl_packet *packet = new pl_packet();
3 packet->setControlInfo( new NetwControlInfo( destination ) );
4 packet->setPayloadArraySize( 1 );
5 packet->setPayload( 0, output );
6 packet->setSrcAddress( getId() );
7 cGate *gOut = gate( "lowerGateOut" );
8 send( packet, gOut );
```

Listing 4.16: Packen und Senden eines Nachrichten-Paketes

Mit einer entsprechenden Sendeverzögerung empfängt das Zielmodul diese Nachricht und kann den Datenwert wieder entnehmen. Daraufhin wird die Simulationsfunktion aufgerufen und der empfangene Wert wird mittels der Sensor-Funktion eingelesen.

Modultypen

Es können zwei Modultypen unterschieden werden:

1. IEEE 802.15.4-Module
2. Module für das Umgebungsmodell

Das Umgebungsmodell, wozu zum Beispiel die Regelstrecke sowie Eingangsgrößen wie Temperatur oder Sonneneinstrahlung gehören, wird mit einem normalen OMNeT++-Modul simuliert. Es ist sozusagen nicht netzwerkfähig, was bei einem Umgebungsmo-

dell aber beabsichtigt ist. Die Kommunikation erfolgt direkt mit den Netzwerkknoten über get- und set-Funktionen, die auf öffentliche Variablen zugreifen. Bei der Regelstrecke kann so ein Messfühler den aktuellen Ausgangswert auslesen und das Stellglied liefert den Eingangswert. Über „Self-Messages“ aktualisiert sich solch ein Modul in regelmäßigen Abständen, indem es die Simulationsfunktion aufruft.

Funktionalitäten der Module

Nicht jedes Netzwerk-Modul ist gleich aufgebaut und funktioniert auch auf die gleiche Art. In Abbildung 4.1 ist ein Regelkreis abgebildet. Der Sollwertgeber hat keinen Eingangsport und das Messglied nur einen Eingang, der per get-Funktion bedient wird. Da OMNeT++ jedoch ein Event-Simulator ist und ein Modul nur aktiv wird, wenn es eine Message empfängt, müssen diese beiden Module anderweitig aktiviert werden.

Die Aktivierung wird stattdessen mittels „Self-Messages“ ausgelöst. Im Falle des Sollwertgebers reicht eine einmalige Aktivierung zur Übermittlung des Sollwertes. Das Messglied muss sich aber in regelmäßigen Abständen aktivieren, um einen Messwert aufzunehmen und zu senden. Energetisch vorteilhaft wäre eine Umsetzung, bei der das Messglied nur sendet, wenn sich der Temperaturwert ändert.

Die beiden anderen Module aus Abbildung 4.1, Regler und Stellglied, brauchen keine Selbstaktivierung. Sie bleiben so lange inaktiv, bis eine Message mit einem Eingangswert eintrifft. Da dies in zeitlich unregelmäßigen Abständen auftreten kann, muss bei Modulen mit kontinuierlichen Blöcken die Zeit *dt* seit dem letzten Aufruf bestimmt werden. Dies kann wie in Listing 4.17 mit einer weiteren Hilfsvariablen realisiert werden.

```
1 dt = simTime().dbl() - last_sim_time;
```

Listing 4.17: Ermittlung der Zeitspanne *dt*

Bei Modulen ohne kontinuierliche Blöcke muss diese Zeit nicht ermittelt werden, da sie für den Ausgangswert nicht relevant ist. Bei solchen Blöcken kann *dt* im Initialisierungsprozess ein fester Wert zugewiesen werden.

Eine weitere Besonderheit sind Blöcke mit mehreren Eingängen. In der *standalone.exe* wird nacheinander zur Eingabe beider Sensorwerte aufgefordert, erst dann wird der Ausgangwert berechnet. Dieses Verhalten kann auch in OMNeT++ umgesetzt werden, indem die Simulationsfunktion nur dann aufgerufen wird, wenn Messages für beide Eingänge eingetroffen sind. Messages können aber nicht gleichzeitig eintreffen, weshalb so eine Zeitdifferenz zwischen beiden Werten entstehen würde.

Da dies nicht verhindert werden kann, ist eine andere Lösung zu bevorzugen. Die Simulationsfunktion wird bei jeder eintreffenden Nachricht aufgerufen und die fehlenden Werte werden von der vorherigen Message übernommen. Im Falle des einfachen Regelkreises aus Abbildung 4.1 ist dies auch sinnvoll, da so der Sollwertgeber nur zu Beginn der Simulation oder gegebenenfalls bei einer Änderung eine Message senden muss. Ab dann ist dem Regler dieser Wert bekannt und er reagiert nur noch auf Messages vom Messglied.

Module mit mehreren Ausgängen sind im Gegensatz dazu unkritisch, da mehrere Messages problemlos nacheinander versendet werden können.

5 Anwendungsbeispiel

Als Anwendungsbeispiel wurde eine verteilte Regelung für eine Raumklima-Regelung gewählt. Klimaregelungen werden in vielen Bereichen, sei es in der Industrie oder in der Hausautomation, eingesetzt. Neben privaten Wohnhäusern werden vor allem öffentliche Einrichtungen beziehungsweise Verkehrsmittel, Werkshallen, Lagerhallen, Gewächshäuser oder Museen klimatisiert. Gründe für die Klimatisierung können ein für den Menschen angenehmes und gesundes Raumklima sein, aber auch praktische Aspekte, wie beispielsweise der Schutz und die Erhaltung von Dokumenten, Waren, technischen Anlagen oder Ausstellungsstücken, wie Gemälden oder historischen Gegenständen.

Bei all diesen Anwendungen geht es hauptsächlich um eine möglichst schnelle und störfeste Regelung der gewünschten Größen. Dabei wird dem Aspekt des Energie-Sparens eine immer größere Bedeutung zugemessen. Jeder Hausbesitzer ist bestrebt seine Energiekosten so gering wie möglich zu halten. In der Industrie spielen die Energiekosten eine noch viel bedeutendere Rolle. Hier besteht in Zukunft ein großes Einspar-Potenzial, da oftmals durch intelligente Nutzung der vorhandenen Ressourcen oder eine gute Steuerung der Stellgrößen viel Energie und somit Geld gespart werden kann.

5.1 Das Modell einer Lagerhalle

Das nun folgende Beispiel behandelt die Klimaregelung einer Lagerhalle. Innerhalb der Halle sollen die Temperatur und die Luftfeuchte geregelt werden können. Dazu wird ein PID-Temperaturregler mit einer Heizung als Stellglied eingesetzt. Zum Regeln der relativen Luftfeuchte wird eine Zwei-Punkt-Regelung verwendet. Das Stellglied der Regelung kann die Raumluft befeuchten und entfeuchten. Dazu wird eine obere und untere Grenze angegeben, ab welcher der Regler eingreifen soll.

Die Außentemperatur und die Sonneneinstrahlung durch die Fenster sind Störgrößen für die Temperaturregelung. Die Sonneneinstrahlung kann aber auch zum Erwärmen

der Lagerhalle genutzt werden. Wenn die Sonneneinstrahlung jedoch zu stark ist, können die Rollläden geschlossen werden, wodurch diese Störgröße eliminiert wird.

In diesem Modell hat eine Temperaturänderung zusätzlich Einfluss auf die Luftfeuchtigkeit in der Halle. Eine Erhöhung der Temperatur senkt die relative Luftfeuchte, eine Abkühlung der Luft bewirkt demnach eine Zunahme der relativen Luftfeuchte. Eine Änderung der Luftfeuchte hat aber keinen Einfluss auf die Temperatur in der Halle.

5.2 Das Szenario

Simuliert werden drei Stunden, in denen die Temperatur und die relative Luftfeuchtigkeit vorgegebenen, veränderlichen Sollwerten folgen soll. Der Startwert der Temperatur ist $T_0 = 0\ ^\circ C$, die relative Luftfeuchte hat einen Startwert von $\varphi_0 = 50\ \%$. Die Außentemperatur und die Sonneneinstrahlung bleiben in dieser Zeit konstant, da selbst eine rasche Änderung dieser Werte wegen der hohen Verzögerungen in der Regelstrecke (Wände und Fenster) durch den PID-Regler problemlos ausgeregelt wird.

Der Sollwert des Temperaturreglers startet bei $T_{s0} = 15\ ^\circ C$, sinkt nach 40 Minuten um drei Kelvin auf $T_{s30} = 12\ ^\circ C$, um nach 2,5 Stunden auf $T_{s150} = 18\ ^\circ C$ anzusteigen. Die relative Luftfeuchtigkeit soll zuerst bei $\varphi_{s0} = 30\ \%$ gehalten werden und nach einer Stunde um 10 Prozentpunkte auf $\varphi_{60} = 40\ \%$ steigen.

5.3 Der Klimaregler

Das vorgegebene Modell wurde unter den gegebenen Voraussetzungen mit SciCos erstellt. Dabei mussten einerseits die späteren Netzwerkknoten modelliert werden andererseits aber auch die Simulationsumgebung, wie zum Beispiel die Regelstrecke.

Da das Beispiel auf keinem realen Schauplatz beruht, der umgesetzt werden sollte, wurde das Modell nur vereinfacht realisiert. Deshalb wird zum Beispiel der Sollwert der Temperatur lediglich als normaler Temperaturwert an den Regler weitergegeben. Es findet keine Analog-Digital-Wandlung wie in einem Mikrocontroller statt. Ebenso gibt das

Stellglied einen Temperaturwert aus und nicht etwa eine Leistung, die erst später in der Regelstrecke in eine Temperatur gewandelt wird. Der Einfluss einer Temperaturänderung auf die relative Luftfeuchtigkeit ist ebenso nur qualitativ umgesetzt worden.

Unter dieser Vorgabe wurde ein Temperaturregler, ein Zweipunkt-Luftfeuchte-Regler und eine Rollladensteuerung entworfen. Diese drei Hauptakteure der verteilten Regelung bestehen aus folgenden Komponenten:

Temperaturregler

- Sollwertgeber
- PID-Regler
- Stellglied
- Thermometer

Luftfeuchteregler

- Sollwertgeber
- Zweipunktregler
- Stellglied
- Hygrometer

Rollladensteuerung

- Steuerung plus Stellglied

Das Umgebungsmodell wird simuliert durch:

- Regelstrecke
- Sonneneinstrahlung
- Außentemperatur

Abbildung 5.1 zeigt das fertige Modell, in dem bereits alle Komponenten in einem Super-Block integriert wurden. Hinter jedem dieser Super-Blöcke steht ein weiteres Modell, bestehend aus den einzelnen Blöcken, die SciCos bereitstellt. Es gibt einen Temperatur-

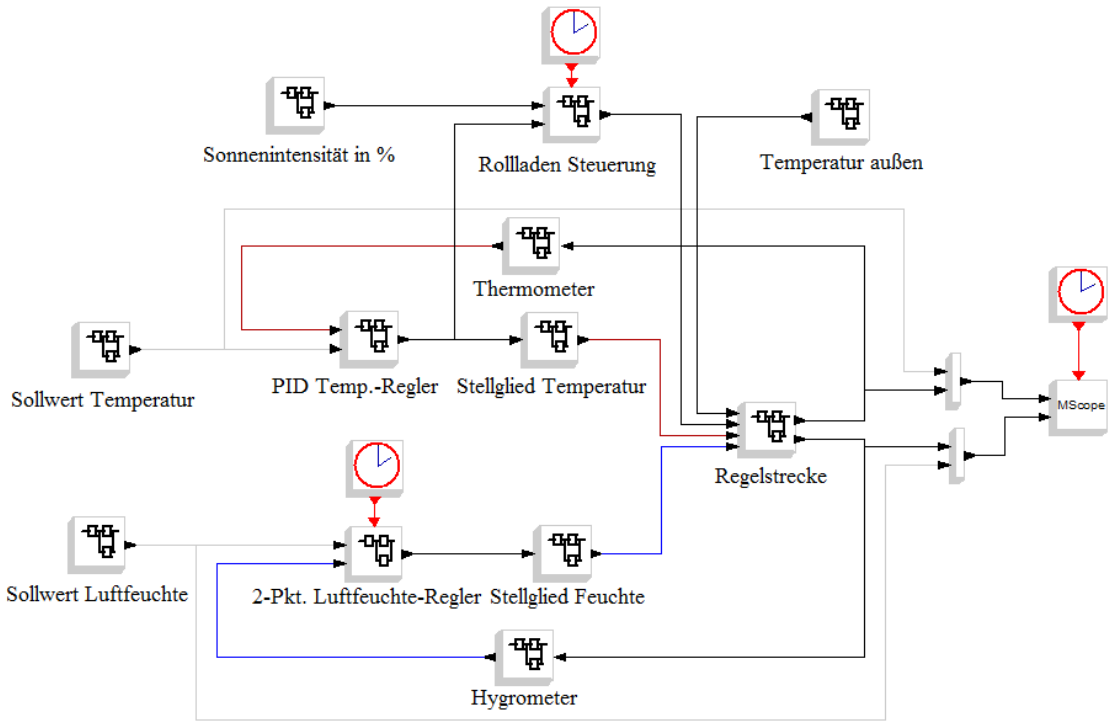


Abbildung 5.1: Verteilte Klimaregelung in SciCos

und einen Luftfeuchtigkeits-Regelkreis. Die Störgrößen Außentemperatur und Sonneneinstrahlung liefern einer gemeinsamen Regelstrecke weitere Eingangsgrößen. Der Sonnenintensität ist eine Rollladensteuerung nachgeschaltet. Diese Steuerung entscheidet anhand der Sonneneinstrahlung und der Stellgröße der Heizung, ob die Rollladen geöffnet oder geschlossen werden müssen. Sie dient somit nur der Temperaturregelung. Die daraus resultierende Helligkeit innerhalb der Halle wurde in dieser Simulation nicht berücksichtigt.

Um auch das Innere eines Super-Blocks zu zeigen, ist die Regelstrecke in Abbildung 5.2 dargestellt. Sie hat vier Eingänge, hinter welchen sich jeweils eine spezielle Übertragungsfunktion beziehungsweise Verzögerung verbirgt, und zwei Ausgänge, einen für die Temperatur und einen für die relative Luftfeuchtigkeit.

Zur Darstellung der einzelnen Werte-Verläufe gibt es einen `Scope`-Block, der jedoch bei der Code-Generierung unberücksichtigt bleibt. OMNeT++ bietet eigene Möglichkeiten zur Darstellung von Kurvenverläufen. In Abbildung 5.3 ist die Ausgabe des `Scope`-Blocks zu sehen. Das obere Diagramm zeigt den Temperaturverlauf und den Tempera-

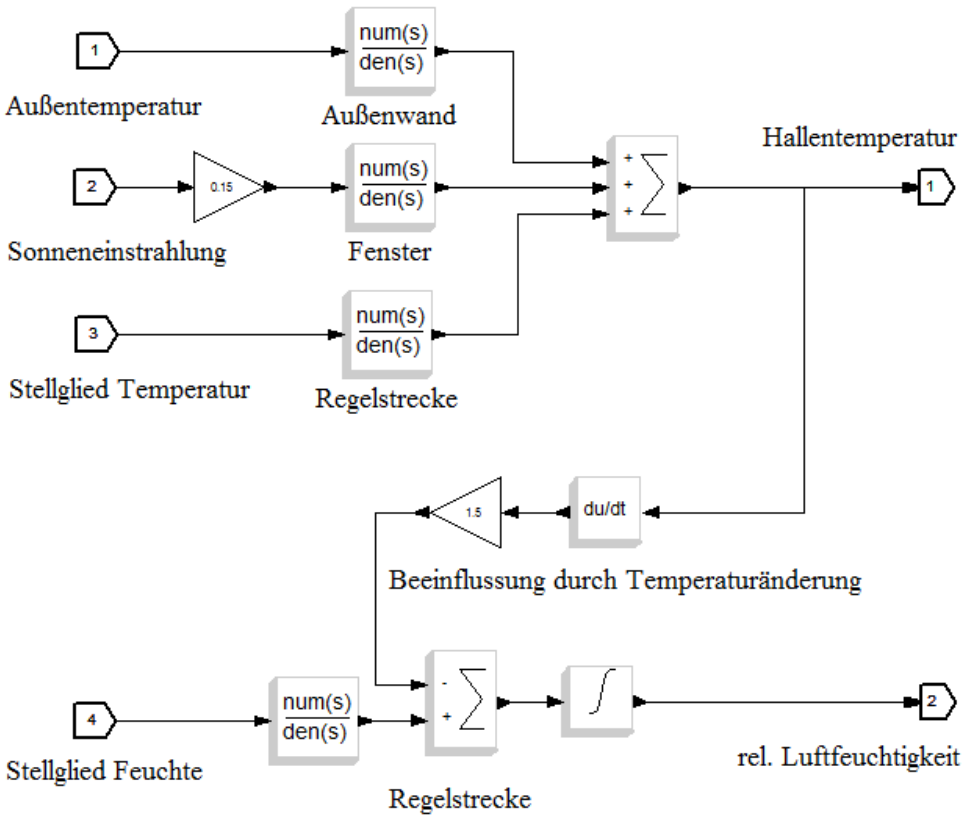


Abbildung 5.2: Regelstrecke der Klimaregelung

tursollwert, das untere die analogen Luftfeuchtigkeitskurven. Die Zeitbasis ist in Sekunden angegeben, da dies auch die Zeitbasis in OMNeT++ ist. So sind die Ergebnisse später am besten zu vergleichen.

Der Temperaturverlauf zeigt, dass der PID-Regler trotz der großen Verzögerungen innerhalb der Regelstrecke schnell und ohne größere Überschwingungen arbeitet. Die kleinen Schwankungen um Sekunde 8000 sind auf ein Schließen des Rollladens zurückzuführen. Diese Störgrößen-Schwankungen kann der Regler problemlos ausregeln.

Der Kurvenverlauf der relativen Luftfeuchtigkeit zeigt ein ähnlich schnelles Regelverhalten. Nur zu Beginn ist ein größeres Überschwingen zu erkennen, da dort der Regler die Luft entfeuchtet und zusätzlich die steigende Temperatur die relative Luftfeuchte sinken lässt. Die teilweise auftretenden Abweichungen im eingeregelten Zustand sind mit der unteren und oberen Schranke des Zweipunktreglers zu begründen. Diese liegen jeweils bei drei Prozentpunkten. Die Beeinflussung der relativen Luftfeuchtigkeit durch

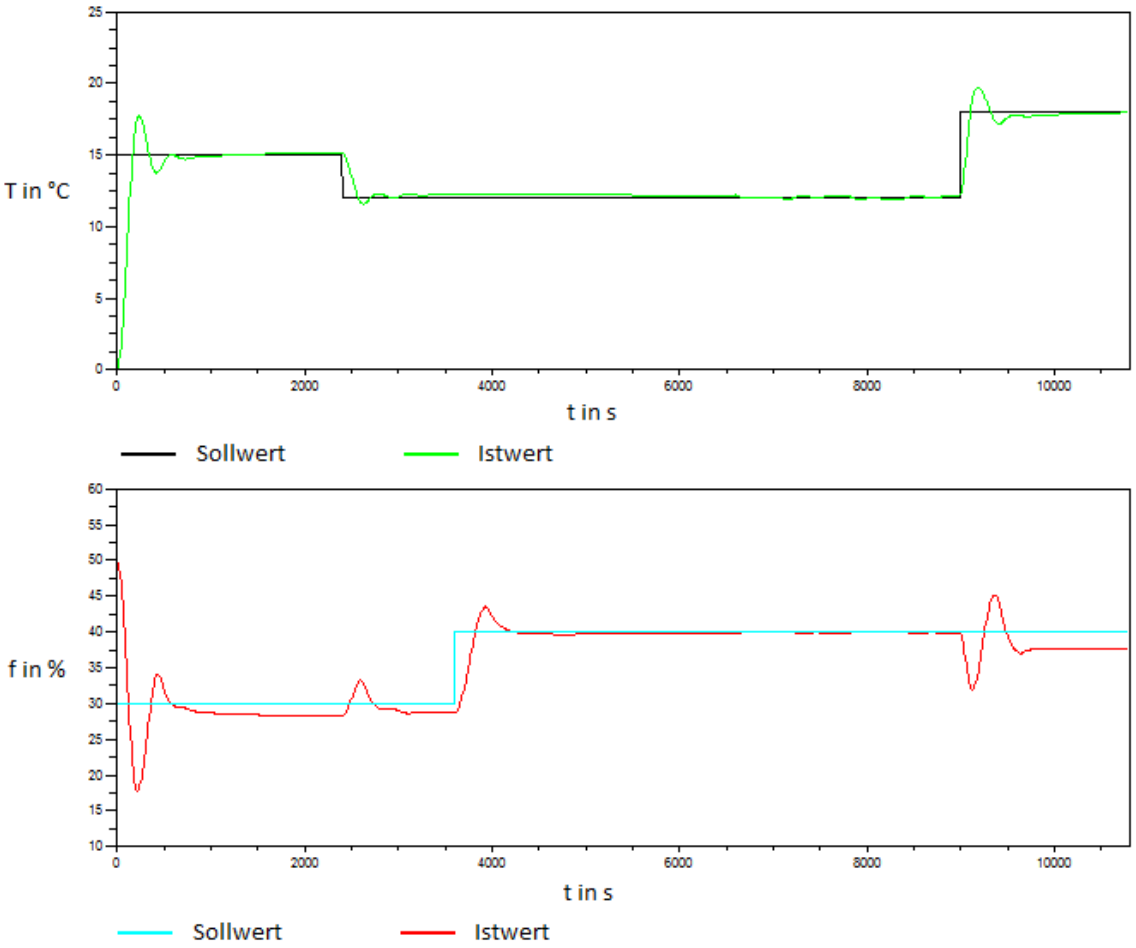


Abbildung 5.3: Kurvenverläufe der Ausgangsgrößen des Klimareglers

eine Temperaturänderung ist um Sekunde 2500 und 9000 gut zu erkennen, doch auch dies wird innerhalb kürzester Zeit ausgeregelt.

5.4 Code-Generierung beim Klimaregler

Beim Zweipunktregler (Abbildung 5.4) wird der `If-Then-Else`-Block verwendet. Damit kann entschieden werden, ob be- oder entfeuchtet werden muss, oder ob keine Reaktion nötig ist, da der Istwert innerhalb der vorgegeben Schranken liegt.

Der `If-Then-Else`-Block ist der einzige, bei dem es zu Problemen bei der Generierung des Codes kommen kann. Bei Verwendung des Code-Generators von SciCos 4.3 war die Ausgabe des Blocks sogar schon bei Verwendung als Standalone-Code fehlerhaft. Das deutet daraufhin, dass dies ein Fehlverhalten des Code-Generators ist.

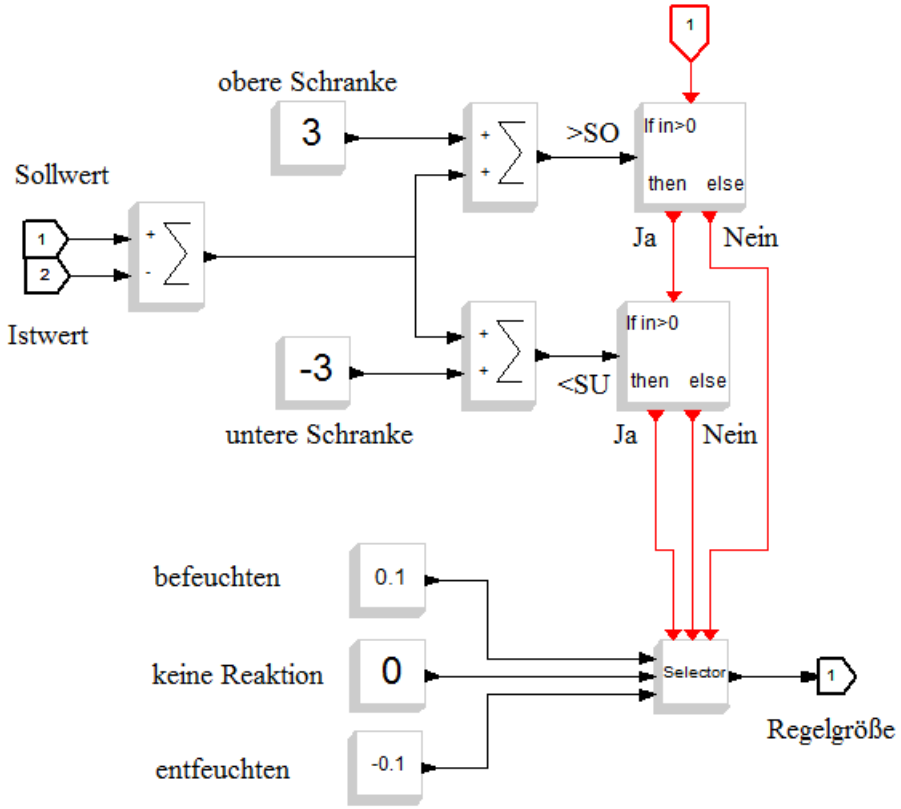


Abbildung 5.4: Zweipunktregler mit `If-Then-Else`-Block

Da sich an dieser Stelle eine Fehlersuche als schwierig erwies, wurde SciCos 4.4b und dessen Code-Generator verwendet, wodurch der Block schließlich sein erwartetes Verhalten zeigte. Deshalb sind der Zweipunktregler und die Rollladensteuerung mit der Beta-Version und die restlichen Blöcke mit Version 4.3 erstellt worden.

Vor Beginn der Code-Generierung sollte das gesamte Modell oder zumindest jeweils die einzelnen Super-Blöcke unter einem anderem Namen gespeichert werden, weil nach der Code-Generierung die einzelnen Super-Blöcke nicht mehr bearbeitet werden können.

Die Code-Generierung verläuft ab da sehr unkompliziert. Der Super-Block wird markiert, um anschließend im Kontextmenü den Eintrag „Code Generation“ auszuwählen (Abbildung 5.5). Im daraufhin erscheinenden Fenster (Abbildung 5.6) kann der Name und der Speicherort des neuen Blocks eingetragen werden. Das Feld „Other object files to link with (if any)“ bleibt leer. Nach dem Bestätigen dieses Dialogfensters beginnt die automatische Code-Generierung. Erscheint auf dem Super-Block anstelle der Grafik

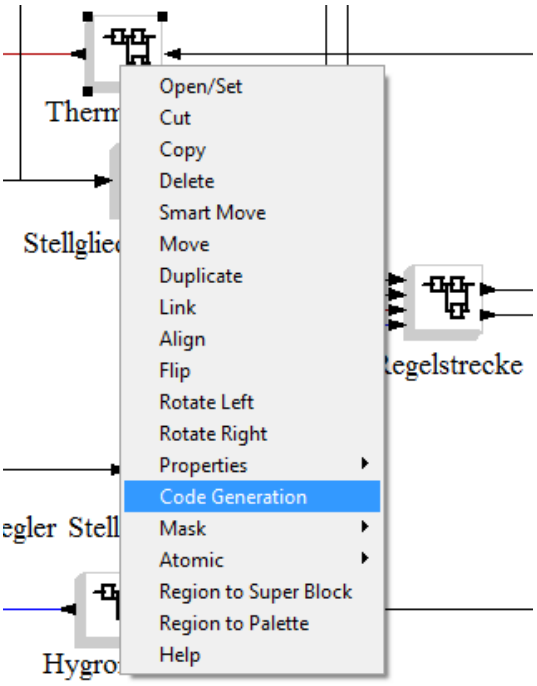


Abbildung 5.5: Kontextmenü eines Super-Blocks

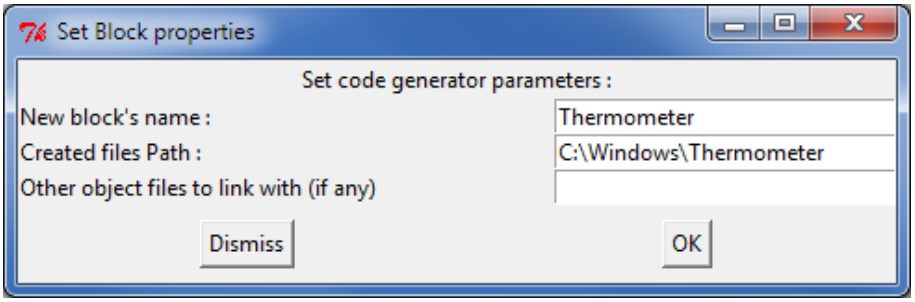


Abbildung 5.6: Dialogfeld zur Code-Generierung

der Name, der zuvor eingegeben wurde, so ist die Code-Generierung abgeschlossen. Im SciCosLab-Hauptfenster wird eine Statusmeldung als Bestätigung ausgegeben. Im angegeben Ordner kann jetzt der Quellcode eingesehen werden.

5.5 Integration in OMNeT++

Integriert wird die verteilte Klimaregelung in einem IEEE 802.15.4-Host (Abbildung 5.7). Die „appl“-Schicht ist die Anwendungsebene, in die der generierte Code die Regelungsanwendung integriert. Die weiteren Netzwerkschichten, die Zusatz-Module sowie die Adressierung und die Message-Übermittlung werden nicht betrachtet. Es muss lediglich

die Netzwerkadresse der Hosts ermittelt werden, um eine Message zu versenden.

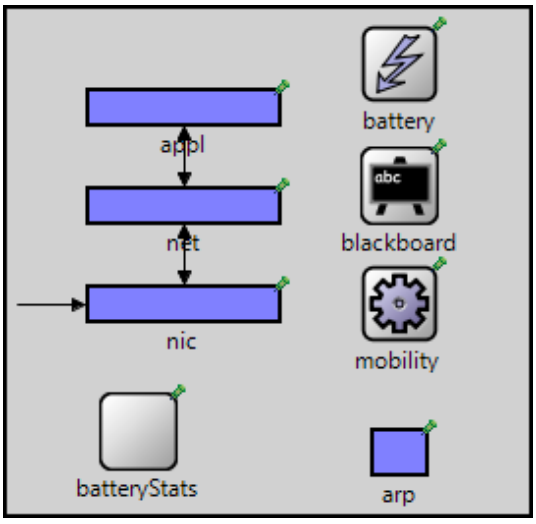


Abbildung 5.7: Host mit seinen Netzwerkschichten und Modulen

Jeder Super-Block benötigt einen eigenen Host, welcher einen IEEE 802.15.4-Knoten darstellt. Die Module des Umgebungsmodells benötigen keinen Host, da sie lediglich ein einfaches OMNeT++-Modul sind und kein Netzwerkknoten.

Jedes dieser Hosts benötigt für die Anwendungsschicht ein weiteres OMNeT++-Modul. Dieses besteht wiederum aus einem NED-File und einer C++-Klasse. Es wird hier von einer MiXiM-Basis-Klasse für die Anwendungsschicht abgeleitet, wodurch die grundlegenden Funktionen „Initialize“, „HandleMessage“ und „HandleSelfMg“ bereitgestellt werden. Die Umgebungsmodelle werden lediglich als „Simple Modul“ erstellt.

Nun wird der Code in diese Klassen integriert und entsprechend den Schritten aus Kapitel 4.4.3 an die OMNeT++-Simulation angepasst. Da ein Routing-Algorithmus nicht integriert ist, müssen die Knoten nah genug beieinander positioniert sein.

Simuliert werden wie in SciCos drei Stunden, was in einer ini-Datei eingetragen werden muss. Der Takt der Sensorknoten beträgt bei der Temperatur und der relativen Luftfeuchte drei Sekunden, da dies ein für die Praxis und die zu regelnden Größen schneller Takt ist. Somit werden auch in diesem Takt die Regler und Stellglieder aufgerufen. Durch den schnellen Aufrufzyklus der Sensorknoten sollte das bestmögliche Regelverhalten erreicht werden.

Eine Funktion zum Aufzeichnen von einzelnen Werten wurde in der Regelstrecke und in den Sollwerten eingebunden. So kann nach Abschluss der Simulation ein Graph aus den aufgenommen Werten erstellt werden. Abbildung 5.8 zeigt den Temperaturverlauf mit Sollwert; Abbildung 5.9 zeigt die Kurven für die relative Luftfeuchtigkeit.

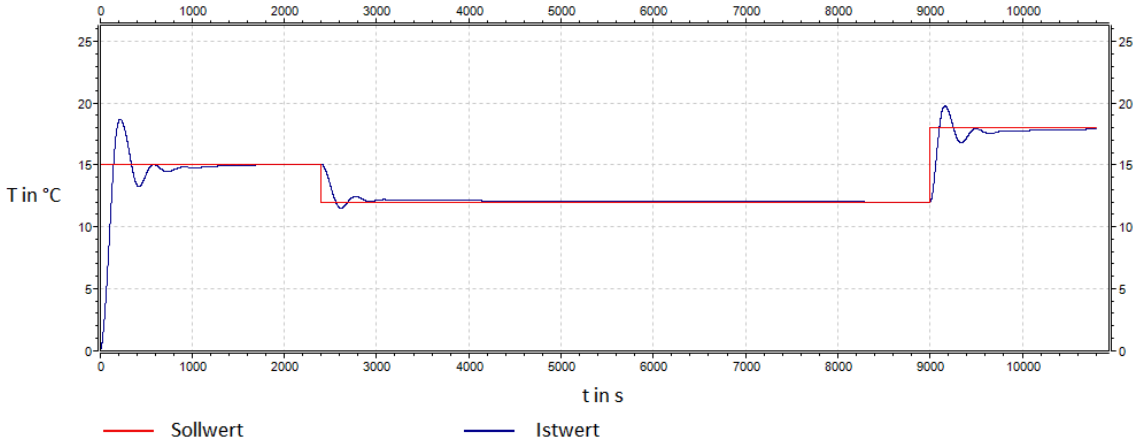


Abbildung 5.8: Kurvenverlauf des Temperaturreglers

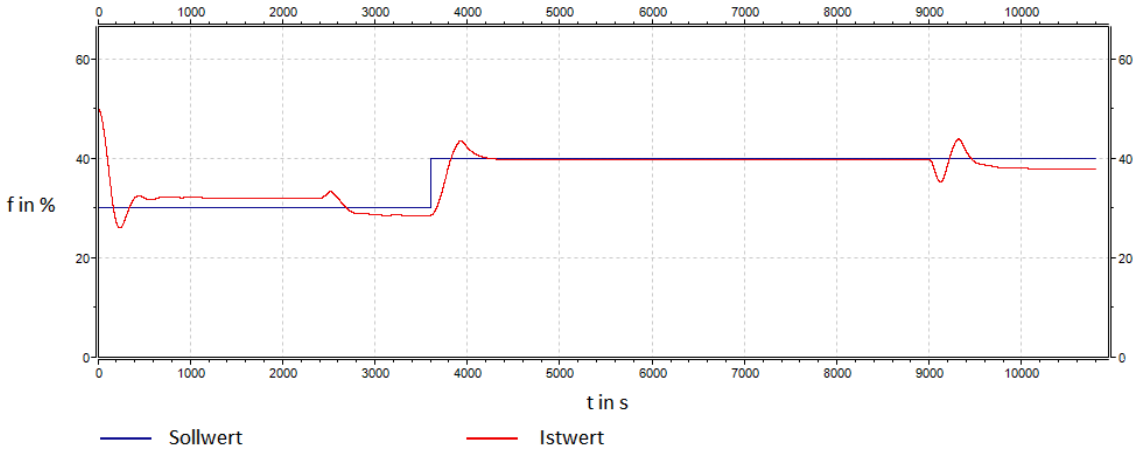


Abbildung 5.9: Kurvenverlauf des Feuchtereglers

5.6 Vergleich der Simulationsergebnisse

Legt man beide Kurven bei gleichem Maßstab übereinander ergibt sich folgender Kurvenverlauf nach Abbildung 5.10: Blau ist jeweils der von OMNeT++ simulierte Regler.

Es ist zu erkennen, dass der Temperaturregler beinah das identische Regelverhalten besitzt, lediglich das Überschwingen zu Beginn fällt um ein Kelvin höher aus. Bei der letzten Temperaturänderung reagiert er sogar etwas schneller.

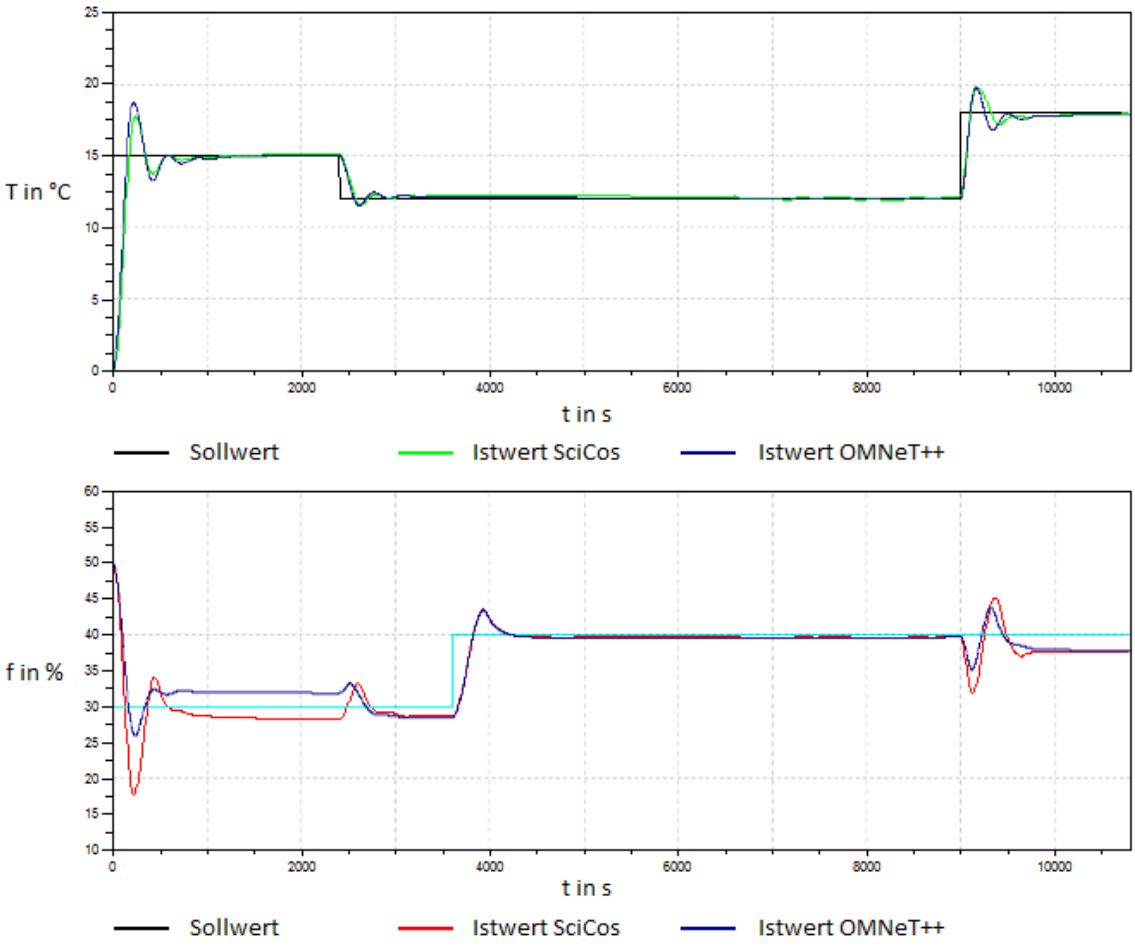


Abbildung 5.10: Vergleich bei einer Abtastung von drei Sekunden

Bei der relativen Luftfeuchte ergibt sich ein sehr erstaunliches Bild. Im Bereich von 3000 bis 9000 Sekunden liegen beide Kurven exakt übereinander, was schlussfolgern lässt, dass der Regler und die Regelstrecke bei der Code-Generierung fehlerfrei übertragen wurden. Die großen Abweichungen im Einschwingvorgang und bei Sekunde 9000 deuten darauf hin, dass der Einfluss einer Temperaturänderung auf die relative Luftfeuchtigkeit in der Regelstrecke in beiden Modellen abweicht. Bei der OMNeT++-Simulation fällt das Überschwingen deutlich geringer aus. Dies liegt aber, wie Abschnitt 3000 bis 9000 deutlich zeigt, nicht am Regler, sondern an einem zufälligen oder systematischen Fehler beim generierten Code der Regelstrecke. Die Abweichungen im eingeschwungenen Zustand um Sekunde 1000 bis 2000 zeigen, dass aufgrund der unteren und oberen Schranke des Zweipunktreglers durchaus ein unterschiedlicher statischer Endwert auftreten kann.

Beim zweiten Kurvenvergleich (Abbildung 5.11) wurden die Sensoren des OMNeT++-Modells mit 30 Sekunden abgetastet. Um dies sinnvoll vergleichen zu können, wurde im SciCos-Modell ebenfalls eine Taktung von 30 Sekunden an den Sensoren eingeführt. So kann die Stabilität des verteilten Netzwerkreglers mit dem SciCos-Modell verglichen werden. Die Ausgabe der OMNeT++-Simulation ist wiederum jeweils durch blaue Kurve dargestellt.

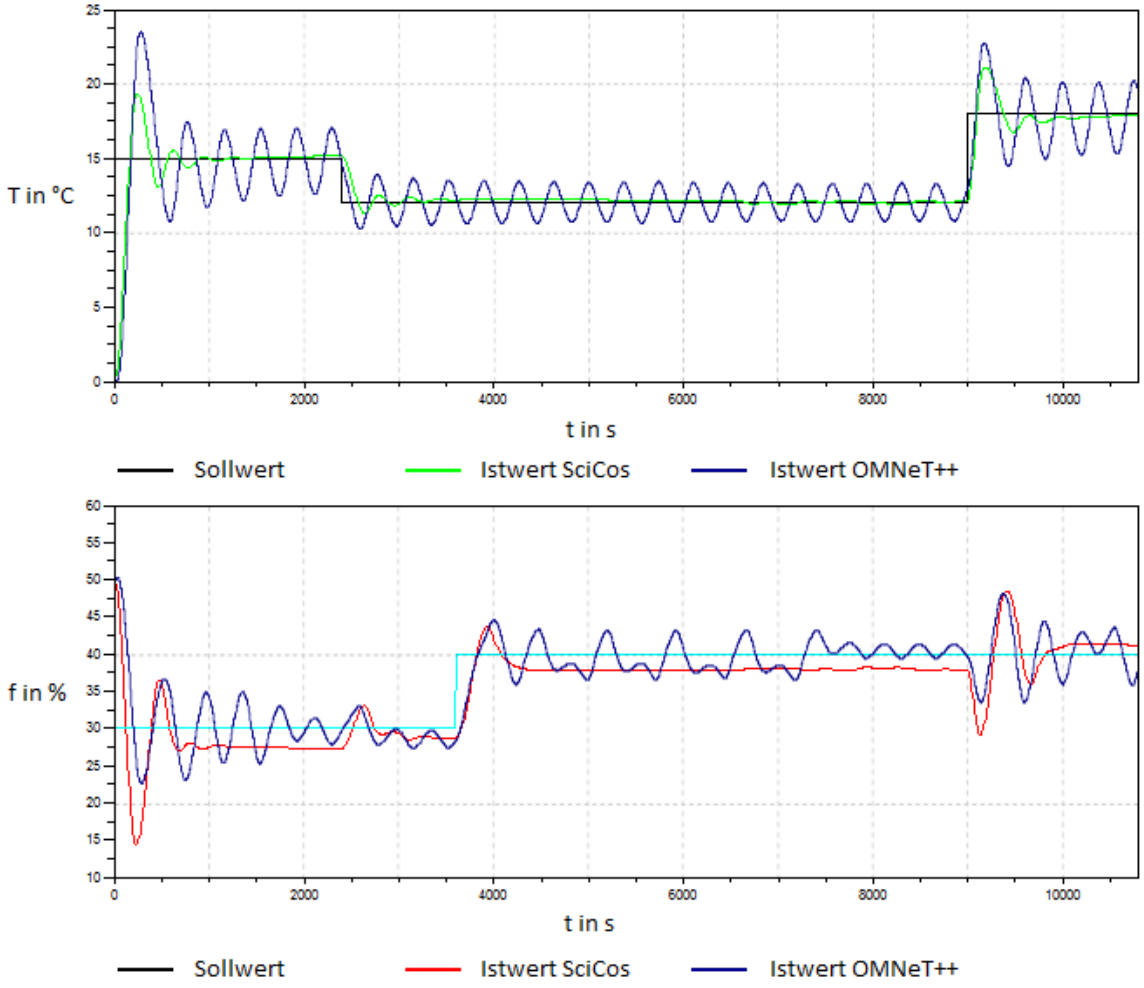


Abbildung 5.11: Vergleich bei einer Abtastung von 30 Sekunden

Während sich die Kurve des SciCos-Modells nur unerheblich verändert hat, ist der OMNeT++-Temperatur-Regler an seiner Stabilitätsgrenze angelangt. Der PID-Regler schwingt stark über und oszilliert danach um den Sollwert.

Der Feuchtigkeitsregler erreicht ebenfalls nicht den eingeschwungenen Zustand. Da es kein kontinuierlicher Regler ist, fällt das Schwingverhalten unregelmäßiger aus.

Die großen Abweichungen bei dieser Abtastung werden eindeutig durch die Verzögerungen im drahtlosen Netzwerk verursacht. Die größeren Totzeiten im Regelkreis senken erheblich die Stabilitätsgrenze. Im SciCos-Modell liegt diese empirisch ermittelt bei zirka 120 Sekunden. In diesem Beispiel wäre, bei gleichbleibendem Stabilitätsverhalten, eine viermal geringere Abtastung möglich.

Es lässt sich festhalten, dass das Regelverhalten bei hinreichend schneller Abtastung relativ exakt nachgebildet werden kann. Die Totzeiten, die durch das Netzwerk entstehen, können dann vernachlässigt werden. Erhöht man aber die Abtastrate, gelangt der Regler im verteilten Netzwerk deutlich schneller an seine Stabilitätsgrenze. Dies geschieht beim kontinuierlichen PID-Regler, wie auch beim Zweipunktregler gleichermaßen.

6 Ausblick

In dieser Diplomarbeit wurde die grundsätzliche Realisierbarkeit der Verknüpfung der Programme SciCos und OMNeT++ über den SciCos-Code-Generator demonstriert. Darüber hinaus sind die umfangreichen Möglichkeiten, die eine solche Simulation bietet aufgezeigt wurden. Eine wie in dieser Arbeit konstruierte Anwendung kann allerdings nicht alle Stärken und Schwächen dieser Co-Simulation sichtbar machen.

Der nächste Schritt innerhalb des SuSAN-Projektes muss es demnach sein, diese Verfahren für echte praktische Anwendung zu nutzen. Erst dann können die Mechanismen entwickelt werden, die den Portierungsprozess nutzbar machen. Somit ergäbe sich eine dringende Notwendigkeit für eine automatische Generierung des OMNeT++-Modells aus dem generierten SciCos-Code.

Es muss aber auch untersucht werden, ob diese Programmverknüpfung wirklich die optimale Lösung ist. OMNeT++ steht als Zielplattform fest, SciCos könnte allerdings ersetzt werden. Es ist eine Open-Source-Software, bei der bei auftretenden Problemen kein echter Support zu erwarten ist. An dieser Stelle wäre Simulink eine gute Alternative, die erprobt werden muss. Die Vorgehensweise wird wahrscheinlich sehr ähnlich der mit SciCos sein.

Aber auch der Verwendung von SciCos stehen keine großen Probleme im Weg. So wäre es empfehlenswert, eine eigene angepasste Blockbibliothek zu erstellen, deren Blöcke in OMNeT++ garantiert funktionieren. Ebenso muss man sich auf eine stabile Version von SciCos festlegen, bei der aus jedem Super-Block garantiert korrekter C-Code gebildet wird.

In OMNeT++ wäre die Erstellung eines Frameworks sinnvoll, welches den generierten C-Code aufnehmen kann. Hier kann eine Systematisierung der verschiedenen Blocktypen, wie Sensor, Aktor, Controller oder das Umgebungsmodell vorgenommen werden, für die in OMNeT++ ein vorgefertigter Host zur Verfügung steht.

Literaturverzeichnis

[1] Wei Zhang, Michael S. Branicky und Stephen M. Phillips: *Stability of Networked Control Systems*; IEEE Control Systems Magazine, Februar 2001.

[2] Martin Andersson, Dan Henriksson, Anton Cervin und Karl-Erik Årzén: *Simulation of Wireless Networked Control Systems*; 44th IEEE Conference on Decision and Control, and the European Control Conference 2005 Sevilla, Spanien, 12. - 15. Dezember 2005.

[3] M Shahidul Hasan, Hongnian Yu und Alison Carrington: *Overview of Wireless Networked Control Systems over Mobile Ad-hoc Network*; 14th International Conference on Automation & Computing, Brunel University, West London, UK, 6. September 2008.

[4] Seite ZigBee FAQ: *http://http://www.meshnetics.com/zigbee-faq*; zuletzt aufgerufen am 30. November 2010.

[5] M Shahidul Hasan, Hongnian Yu und Alison Griffiths und T C Yang: *Co-simulation framework for Networked Control Systems over multi-hop mobile ad-hoc networks*; The International Federation of Automatic Control, Seoul, Korea, 6. - 11. Juli 2008.

[6] Seite 'Simulink'. In: Wikipedia, Die freie Enzyklopädie: *http://de.wikipedia.org/wiki/Simulink* zuletzt aufgerufen am 1. September 2010.

[7] Seite 'Modelica'. In: Wikipedia, Die freie Enzyklopädie: *http://de.wikipedia.org/wiki/Modelica*; zuletzt aufgerufen am 10. Dezember 2010.

[8] Seite 'ScicosLab' In: Wikipedia, Die freie Enzyklopädie: *http://de.wikipedia.org/wiki/Scilab*; zuletzt aufgerufen am 10. Dezember 2010.

[9] Network Simulator 2: *http://nsnam.isi.edu/nsnam/index.php/User_Information*; zuletzt aufgerufen am 14. Dezember 2010.

[10] OPNET Modeler: *http://www.opnet.com/solutions/network_rd/modeler.html*; zuletzt aufgerufen am 14 Dezember 2010.

[11] OMNeT++: *http://www.omnetpp.org/home/what-is-omnet*; zuletzt aufgerufen am 29. Dezember 2010.

[12] Fu Jing-qi, Xu Cai-xian, Kan Bao-dong und Wang Hai-kuan: *Design and Simulation of Control Algorithms for WINCS*; School of Mechanical and Electrical Engineering and Automation, Shanghai University, Shanghai, China, 2009.

[13] TrueTime Handbuch:

http://www.control.lth.se/database/publications/article.pike?artkey=ohl%2b07tt; zuletzt aufgerufen am 29. Dezember 2010.

[14] TrueTime: *http://www.control.lth.se/truetime/*; zuletzt aufgerufen am 29. Dezember 2010.

[15] Tuomo Kohtamäki, Mikael Pohjola, Jenna Brand und Lasse M. Eriksson: *PiccSIM Toolchain - Design, Simulation and Automatic Implementation of Wireless Networked Control Systems*; 2009 IEEE International Conference on Networking, Sensing and Control, Okayama, Japan, 26. - 29. März 2009.

[16] M.S. Hasan, H. Yu, A. Carrington und T.C. Yang: *Co-simulation of wireless networked control systems over mobile ad hoc network using SIMULINK and OPNET*; IET Communication, 2009, Volume 3, Issue 8, page 1297-1310.

[17] A. Al-hammouri, V. Liberatore, H. Al-omari, Z. Al-qudah, M. Branicky und D. Agrawal: *A co-simulation platform for actuator networks*; ACM Conference on Embedded Networked Sensor Systems, 2007.

[18] Seite 'UNIX' In: Wikipedia, Die freie Enzyklopädie: *http://de.wikipedia.org/wiki/Unix#Merkmale*; zuletzt aufgerufen am 13. Dezember 2010.

[19] SciCos: *http://www-rocq.inria.fr/scicos/*; zuletzt aufgerufen am 13. Dezember 2010.

[20] Stephen L. Campbell, Jean-Philippe Chancelier und Ramine Nikoukhah: *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*; Springer Verlag, Second Edition, 2010.

[21] OMNeT++ Handbuch: *http://www.omnetpp.org/doc/omnetpp41/manual/usman.html*; zuletzt aufgerufen am 10. September 2010.

[22] MiXiM: *http://mixim.sourceforge.net/*; zuletzt aufgerufen am 9. Dezember 2010.

[23] Jan Lunze: *Regelungstechnik 2; Mehrgrößensysteme, Digitale Regelung*; Springer Verlag; 6. neu bearbeitete Auflage, 2010.

[24] Shahin Farahani: *ZigBee Wireless Networks and Transceivers*; Butterworth Heinemann; Auflage: Pap/Com, 2008.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, 3. Januar 2011